QCFractal Documentation

Release v0.15.5

The QCArchive Development Team

Mar 21, 2021

GETTING STARTED

A platform to compute, store, organize, and share large-scale quantum chemistry data.

QCFractal emphasizes the following virtues:

- **Organize:** Large sets of computations are organized into Collections for easy reference and manipulation.
- **Reproducibility:** All steps of commonly used pipelines are elucidated in the input without additional human intervention.
- **History:** Organize all computations ever executed in a research group in an easily indexable manner, never lose data again!
- **Accessibility:** Easily share quantum chemistry data with colleagues or the community through accessibility settings.
- **Elasticity:** Scale from a single laptop to a dozen simultaneous supercomputers.
- **Transferability:** Run many different quantum chemistry, semiempirical, or force field programs with a single unified interface.

CHAPTER

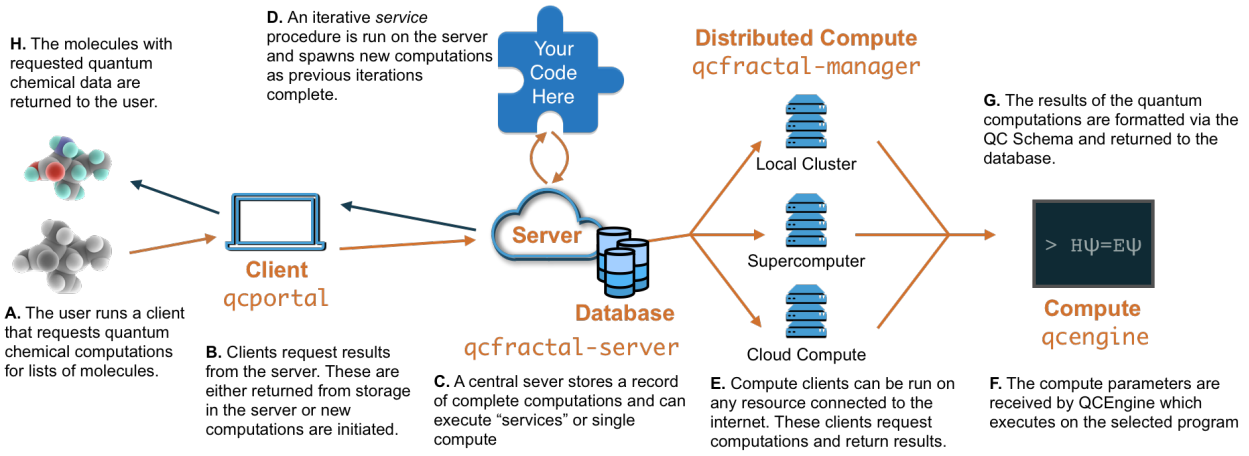
ONE

QCFRATAL WITHIN THE QCARCHIVE STACK

Within the QCArchive stack, QCFractal is responsible for:

- Accepting requests for calculations, submitted from a client such as QCPortal.
- Running these calculations on disparate compute resources through Quantum Chemistry programs and procedures supported by QCEngine.
- Storing and organizing the results of these calculations in a database.
- Providing access to stored results to clients such as QCPortal.

The diagram below illustrates a representative data flow:



CHAPTER

TWO

PIPELINES

QCFractal supports several high-throughput pipelines:

- Ensembles of single point quantum chemistry computations.
- Procedures such as geometry optimization, finite different gradients and Hessians, and complete basis set extrapolations.
- Complex scenarios such as the OpenFF torsion scan workflow.
- User-defined procedures.

CHAPTER

THREE

DATA SHARING

QCFractal allows for the creation of a single data repository for a group with varying permission levels to allow control of who can access your data or submit new tasks to your compute network. Hook into the central MolSSI repository or create your own!

CHAPTER

FOUR

SCALES FROM LAPTOPS TO SUPERCOMPUTERS

QCFractal is build to be elastic, scaling from a single researcher on a laptop to large multi-PI groups on dozens of different supercomputers. QCFractal provides a central location to marshal and distribute data or computation. QCFractal can switch between a variety of computational queue backends such as:

- Parsl - High-performance workflows with backend support for common schedulers, supercomputers, and cloud compute.
- Dask - A graph-based workflow engine for laptops and small clusters.
- Fireworks - A asynchronous Mongo-based distributed queuing system.

Additional backends such as BOINC, RADICAL Pilot, Kubernetes, and Balsam are under active consideration. Contact us if you are interested in one of these use-cases.

CHAPTER

FIVE

INDEX

Getting Started

5.1 Install QCFractal

You can install QCFractal with `conda` (recommended) or with `pip` (with some caveats).

The below commands install QCFractal and its required dependencies, but *not* any of the quantum chemistry codes nor the software to run *Queue Managers*. This is done to avoid requiring *all* software which *can* interface with QCFractal, and instead requires the user to obtain the software they individually *require*.

5.1.1 Conda

You can install QCFractal using conda:

```
>>> conda install qcfractal -c conda-forge
```

This installs QCFractal and its dependencies. The QCFractal package is maintained on the conda-forge channel.

Conda Pre-Created Environments

QCFractal can also be installed through pre-configured environments you can pull through our Conda Channel:

```
>>> conda env create qcarchive/{environment name}
>>> conda activate {environment name}
```

The environments are created from the YAML files hosted on the Anaconda Cloud, which then need to be activated to use. You can find all of the environments here.

If you want to use a different name than the environment file, you can add a `-n {custom name}` flag to the `conda env` command.

The environments must be installed as new environments and cannot be installed into existing ones.

The environments are designed to provide pre-built environments which include additional programs beyond QCFractal itself which are designed for use in production or practical experimentation. For instance, the `qcf-manager-openff` environment also installs a couple quantum chemistry programs, a distributed compute *Queue Adapter*, and a service which QCFractal can run. This environment can be deployed for immediate use on a remote compute site (e.g. a cluster) and connect to a QCFractal instance to consume compute tasks.

5.1.2 Pip

Warning: Installing QCFractal from PyPi/`pip` requires an existing PostgreSQL installation on your computer. Whether that be through a native install on your device (e.g. managed clusters), a direct installer, `yum` install, a `conda` install, or otherwise; it must be installed first or the `Psycopg2` package will complain about missing the `pg_config`. Installation of PostgreSQL manually is beyond the scope of these instructions, so we recommend either using a *Conda install of QCFractal* or contacting your systems administrator.

If you have PosgreSQL installed already, you can also install QCFractal using `pip`:

```
>>> pip install qcfractal
```

5.1.3 Test the Installation

Note: There are several optional packages QCFractal can interface with for additional features such as visualization, *Queue Adapters*, and services. These are not installed by default and so you can expect many of the tests will be marked with `skip` or `s`.

You can test to make sure that Fractal is installed correctly by first installing `pytest`.

From `conda`:

```
>>> conda install pytest -c conda-forge
```

From `pip`:

```
>>> pip install pytest
```

Then, run the following command:

```
>>> pytest --pyargs qcfractal
```

QCFractal ships with a small testing plugin which should be automatically detected and gives you access to the `--runslow` and `--runexamples` PyTest CLI flags. The `--runslow` flag tells the testing suite to run any test the developers think are a bit more time consuming than the others. Without this flag, you will see many tests (such as those for the CLI) skipped.

5.1.4 Developing from Source

If you are a developer and want to make contributions QCFractal, you can access the source code from github.

5.2 Example

This tutorial will go over general QCFractal usage to give a feel for the ecosystem. In this tutorial, we employ Snowflake, a simple QCFractal stack which runs on a local machine for demonstration and exploration purposes.

5.2.1 Installation

To begin this quickstart tutorial, first install the QCArchive Snowflake environment from conda:

```
conda env create qcarchive/qcfractal-snowflake -n snowflake
conda activate snowflake
```

If you have a pre-existing environment with `qcfractal`, ensure that `rdkit` and `geometric` are installed from the `conda-forge` channel and `psi4` and `dftd3` from the `psi4` channel.

5.2.2 Importing QCFractal

First let us import two items from the ecosystem:

- FractalSnowflakeHandler - This is a FractalServer that is temporary and is used for trying out new things.
- `qcfractal.interface` is the QCPortal module, but if using QCFractal it is best to import it locally.

Typically we alias `qcportal` as `ptl`. We will do the same for `qcfractal.interface` so that the code can be used anywhere.

[1]:
```
from qcfractal import FractalSnowflakeHandler
import qcfractal.interface as ptl
```

We can now build a temporary server which acts just like a normal server, but we have a bit more direct control of it.

Warning! All data is lost when this notebook shuts down! This is for demonstration purposes only! For information about how to setup a permanent QCFractal server, see the *Setup Quickstart Guide*.

[2]:
```
server = FractalSnowflakeHandler()
server
```

[2]:
```
FractalSnowflakeHandler(name='db_4dd4a305_1692_4f29_ae5a_ac4c8bcb1002' uri='https://
↪localhost:60885')
```

We can then build a typical FractalClient to automatically connect to this server using the client() helper command. Note that the server names and addresses are identical in both the server and client.

[3]:
```
client = server.client()
client
```

[3]:
```
FractalClient(server_name='FractalSnowFlake_db_4dd4a', address='https://localhost:
↪60885/', username='None')
```

5.2.3 Adding and Querying data

A server starts with no data, so let's add some! We can do this by adding a water molecule at a poor geometry from XYZ coordinates. Note that all internal QCFractal values are stored and used in atomic units; whereas, the standard Molecule.from_data() assumes an input of Angstroms. We can switch this back to Bohr by adding a `units` command in the text string.

[4]:
```
mol = ptl.Molecule.from_data("""
O 0 0 0
H 0 0 2
H 0 2 0
units bohr
""")
mol
```

```
_ColormakerRegistry()
```

```
NGLWidget()
```

We can then measure various aspects of this molecule to determine its shape. Note that the `measure` command will provide a distance, angle, or dihedral depending if 2, 3, or 4 indices are passed in.

This molecule is quite far from optimal, so let's run an geometry optimization!

[5]:
```
print(mol.measure([0, 1]))
print(mol.measure([1, 0, 2]))
```

```
2.0
90.0
```

5.2.4 Evaluating a Geometry Optimization

We originally installed `psi4` and `geometric`, so we can use these programs to perform a geometry optimization. In QCFractal, we call a geometry optimization a `procedure`, where `procedure` is a generic term for a higher level operation that will run multiple individual quantum chemistry energy, gradient, or Hessian evaluations. Other `procedure` examples are finite-difference computations, n-body computations, and torsiondrives.

We provide a JSON-like input to the client.add_procedure() command to specify the method, basis, and program to be used. The `qc_spec` field is used in all procedures to determine the underlying quantum chemistry method behind the individual procedure. In this way, we can use any program or method that returns a energy or gradient quantity to run our geometry optimization! (See also add_compute().)

[6]:
```
spec = {
    "keywords": None,
    "qc_spec": {
        "driver": "gradient",
        "method": "b3lyp",
        "basis": "6-31g",
        "program": "psi4"
    },
}

# Ask the server to compute a new computation
r = client.add_procedure("optimization", "geometric", spec, [mol])
print(r)
print(r.ids)
```

```
ComputeResponse(nsubmitted=1 nexisting=0)
['1']
```

We can see that we submitted a single task to be evaluated and the server has not seen this particular procedure before. The `ids` field returns the unique `id` of the procedure. Different procedures will always have a unique `id`, while identical procedures will always return the same `id`. We can submit the same procedure again to see this effect:

[7]:
```
r2 = client.add_procedure("optimization", "geometric", spec, [mol])
print(r)
print(r.ids)
```

```
ComputeResponse(nsubmitted=1 nexisting=0)
['1']
```

5.2.5 Querying Procedures

Once a task is submitted, it will be placed in the compute queue and evaluated. In this particular case the FractalSnowflakeHandler uses your local hardware to evaluate these jobs. We recommend avoiding large tasks!

In general, the server can handle anywhere between laptop-scale resources to many hundreds of thousands of concurrent cores at many physical locations. The amount of resources to connect is up to you and the amount of compute that you require.

Since we did submit a very small job it is likely complete by now. Let us query this procedure from the server using its `id` like so:

[15]:
```
proc = client.query_procedures(id=r.ids)[0]
proc
```

[15]:
```
<OptimizationRecord(id='1' status='COMPLETE')>
```

This OptimizationRecord object has many different fields attached to it so that all quantities involved in the computation can be explored. For this example, let us pull the final molecule (optimized structure) and inspect the physical dimensions.

Note: if the status does not say `COMPLETE`, these fields will not be available. Try querying the procedure again in a few seconds to see if the task completed in the background.

[16]:
```
final_mol = proc.get_final_molecule()
```

[17]:
```
print(final_mol.measure([0, 1]))
print(final_mol.measure([1, 0, 2]))
final_mol
```

```
1.844111236246719
108.30246679333517
```

```
NGLWidget()
```

This water molecule has bond length and angle dimensions much closer to expected values. We can also plot the optimization history to see how each step in the geometry optimization affected the results. Though the chart is not too impressive for this simple molecule, it is hopefully illuminating and is available for any geometry optimization ever completed.

[18]:
```
proc.show_history()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

5.2.6 Collections

Submitting individual procedures or single quantum chemistry tasks is not typically done as it becomes hard to track individual tasks. To help resolve this, `Collections` are different ways of organizing standard computations so that many tasks can be referenced in a more human-friendly way. In this particular case, we will be exploring an intermolecular potential dataset.

To begin, we will create a new dataset and add a few intermolecular interactions to it.

[19]:
```
ds = ptl.collections.ReactionDataset("My IE Dataset", ds_type="ie", client=client,␣
↪default_program="psi4")
```

We can construct a water dimer that has fragments used in the intermolecular computation with the `--` divider. A single water molecule with ghost atoms can be extracted like so:

[20]:
```
water_dimer = ptl.Molecule.from_data("""
O 0.000000 0.000000  0.000000
H 0.758602 0.000000  0.504284
H 0.260455 0.000000 -0.872893
--
O 3.000000 0.500000  0.000000
H 3.758602 0.500000  0.504284
H 3.260455 0.500000 -0.872893
""")

water_dimer.get_fragment(0, 1)
```

```
NGLWidget()
```

Many molecular entries can be added to this dataset where each is entry is a given intermolecular complex that is given a unique name. In addition, the `add_ie_rxn` method to can automatically fragment molecules.

[21]:
```
ds.add_ie_rxn("water dimer", water_dimer)
ds.add_ie_rxn("helium dimer", """
He 0 0 -3
--
He 0 0 3
""")
```

[21]:
```
ReactionEntry(attributes={}, reaction_results={'default': {}}, name='helium dimer',␣
↪stoichiometry={'default1': {'b3855c64e9f61158f5e449e2df7b79bf1fa599d7': 2.0}, 'cp1':
↪ {'e925ae0ef797c65c359e9c8029f0342df495d116': 2.0}, 'default': {
↪'a0d757c65af76f309b2685f31f63b3da781d0ba7': 1.0}, 'cp': {
↪'a0d757c65af76f309b2685f31f63b3da781d0ba7': 1.0}}, extras={})
```

Once the Collection is created it can be saved to the server so that it can always be retrived at a future date

[22]:
```
ds.save()
```

[22]:
```
'1'
```

The client can list all Collections currently on the server and retrive collections to be manipulated:

[23]:
```
client.list_collections()
```

[23]:
```
                              tagline
collection      name
ReactionDataset My IE Dataset    None
```

[24]:
```
ds = client.get_collection("ReactionDataset", "My IE Dataset")
ds
```

[24]:
```
<ReactionDataset(name=`My IE Dataset`, id='1', client='https://localhost:60885/') >
```

5.2.7 Computing with collections

Computational methods can be applied to all of the reactions in the dataset with just a few simple lines:

[25]:
```
ds.compute("B3LYP-D3", "def2-SVP")
```

[25]:
```
<ComputeResponse(nsubmitted=10 nexisting=0)>
```

By default this collection evaluates the non-counterpoise corrected interaction energy which typically requires three computations per entry (the complex and each monomer). In this case we compute the B3LYP and -D3 additive correction separately, nominally 12 total computations. However the collection is smart enough to understand that each Helium monomer is identical and does not need to be computed twice, reducing the total number of computations to 10 as shown here. We can continue to compute additional methods. Again, this is being evaluated on your computer! Be careful of the compute requirements.

[26]:
```
ds.compute("PBE-D3", "def2-SVP")
```

[26]:
```
<ComputeResponse(nsubmitted=10 nexisting=0)>
```

A list of all methods that have been computed for this dataset can also be shown:

[28]:
```
ds.list_values()
```

[28]:
```
                                                stoichiometry  \
native driver program method   basis    keywords
True   energy psi4    b3lyp    def2-svp None           default
                     b3lyp-d3 def2-svp None           default
                     pbe      def2-svp None           default
                     pbe-d3   def2-svp None           default


                                                            name
native driver program method   basis    keywords
True   energy psi4    b3lyp    def2-svp None         B3LYP/def2-svp
                     b3lyp-d3 def2-svp None      B3LYP-D3/def2-svp
                     pbe      def2-svp None           PBE/def2-svp
                     pbe-d3   def2-svp None        PBE-D3/def2-svp
```

The above only shows what has been computed and does not pull this data from the server to your computer. To do so, the `get_history` command can be used:

[32]:
```
print(f"DataFrame units: {ds.units}")
ds.get_values()
```

```
DataFrame units: kcal / mol
```

[32]:
```
             PBE/def2-svp B3LYP/def2-svp PBE-D3/def2-svp B3LYP-D3/def2-svp
water dimer      -5.11552       -4.75187        -5.63188          -5.57767
helium dimer  0.000684115    0.000674748     0.000207208        0.00017274
```

You can also visualize results and more!

```
[33]: ds.visualize(["B3LYP-D3", "PBE-D3"], "def2-SVP", bench="B3LYP/def2-svp", kind="violin
      ↪")
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

This is only the very beginning of what you can do with QCFractal! Explore the documentation to learn more capabilities. In particular, see the *next section* for a quickstart guide on how to set up QCFractal in production.

If you like the project, consider starring us on GitHub. If you have any questions, join our Slack channel.

5.3 Setup Overview

QCFractal comprises two components:

1. The *Server* (`qcfractal-server`), which accepts compute and data queries and maintains a database of *tasks* and results. The *Server* should be run continuously on a persistent machine.
2. One or more *Managers* (`qcfractal-manager`). The *Managers* pull work from the *Server*, use attached compute resources to complete the work, and report results back to the server. *Managers* may be turned off and on at any time. *Managers* connect to compute resources through *Adapters*.

In the *Quickstart Tutorial*, the above components were combined within a python environment using `FractalSnowflake`. In general, the *Server* and *Manager(s)* are run separately in different processes, often on different machines. For detailed information about the relationship between *Server* and *Manager*, see *Fractal Queue Managers*.

5.3.1 Common Use Cases

The table below lists some common use cases for QCFractal:

Use case	`qcfractal-server` location	`qcfractal-manager` location	Recommended manager
Demonstration/Exploration	Snowflake	Snowflake	Snowflake
Single Workstation	Local	Local	Pool
Private Cluster	Head node	Head node	Parsl
Shared Cluster/Supercomputer	Personal server, head node (if permitted)	Head node	Parsl
Multiple Clusters	Personal server	Head node of each cluster	Parsl
Cloud Compute	Personal server or cloud instance	Docker container	Pool

QCFractal is highly adaptable and is not limited to the above use cases. For example, it possible to mix local, cluster, supercomputer, and cloud *Managers* simultaneously. In addition, a cloud instance may provide a good option for running `qcfractal-server` when a persistent web-exposed server is not otherwise available.

5.3.2 Quickstart Setups

This section presents quickstart setup guides for the above common use cases. The guides assume that QCFractal has been installed (see *Install QCFractal*). General guides are also available:

- *Server Setup*
- *Manager Setup*

Single Workstation

This quickstart guide addresses QCFractal setup on a single computer which will be used for the *Server*, *Manager*, user client, and compute. On the workstation, initialize the *Server*:

```
qcfractal-server init
```

Next, start the *Server* and ProcessPoolExecutor *Manager*:

```
nohup qcfractal-server start --local-manager 1 &
```

The second command starts `qcfractal-server` in the background. It also starts one *Worker* which will pull *tasks* from the *Server* and run them.

Test if everything is setup by running a Hartree-Fock calculation on a single hydrogen molecule, as in the *Example* (note this requires `psi4`):

```
python

>>> import qcfractal.interface as ptl

# Note that server TLS verification is turned off (verify=False) since all components␣
↪are run locally.
>>> client = ptl.FractalClient(address="localhost:7777", verify=False)
>>> mol = ptl.Molecule(symbols=["H", "H"], geometry=[0, 0, 0, 0, 5, 0])
>>> mol_id = client.add_molecules([mol])[0]
>>> r = client.add_compute("psi4", "HF", "STO-3G", "energy", None, [mol_id])

# Wait a minute for the job to complete
>>> proc = client.query_procedures(id=r.ids)[0]
>>> print(proc)
<ResultRecord(id='0' status='COMPLETE')>
>>> print(proc.properties.scf_total_energy)
-0.6865598095254312
```

Private Cluster

This quickstart guide addresses QCFractal setup on a private cluster comprising a head node and compute nodes, with a *Scheduler* such as SLURM, PBS, or Torque. This guide requires Parsl which may be installed with `pip` or `conda`.

Begin by initializing the *Server* on the cluster head node:

```
qcfractal-server init
```

Next, start the *Server* in the background:

```
nohup qcfractal-server start &
```

The *Manager* must be configured before use. Create a configuration file (e.g. in `~/.qca/qcfractal/my_manager.yaml`) based on the following template:

```
common:
 adapter: parsl
 tasks_per_worker: 1
 cores_per_worker: 6
 memory_per_worker: 64
 max_workers: 5
 scratch_directory: "$TMPDIR"

cluster:
 node_exclusivity: True
 scheduler: slurm

parsl:
 provider:
  partition: CLUSTER
  cmd_timeout: 30
```

You may need to modify these values to match the particulars of your cluster. In particular:

- The `scheduler` and `partition` options should be set to match the details of your *Scheduler* (e.g. SLURM, PBS, Torque).
- Options related to *Workers* should be set appropriately for the compute node on your cluster. Note that Parsl requires that full nodes be allocated to each *Worker* (i.e. `node_exclusivity: True`).

For more information on *Manager* configuration, see *Fractal Queue Managers* and *Queue Manager Example YAML Files*.

Finally, start the *Manager* in the background on the cluster head node:

```
nohup qcfractal-manager --config-file <path to config YAML> --verify=False &
```

Note that TLS certificate verification is disabled (`--verify=False`) because the *Manager* and *Server* are both run on the head node.

Test if everything is setup by running a Hartree-Fock calculation on a single hydrogen molecule, as in the *Example* (note this requires `psi4`):

```
python

>>> import qcfractal.interface as ptl

# Note that server TLS verification is turned off (verify=False) since all components␣
↪are run locally.
>>> client = ptl.FractalClient(address="localhost:7777", verify=False)
>>> mol = ptl.Molecule(symbols=["H", "H"], geometry=[0, 0, 0, 0, 5, 0])
>>> mol_id = client.add_molecules([mol])[0]
>>> r = client.add_compute("psi4", "HF", "STO-3G", "energy", None, [mol_id])

# Wait a minute for the job to complete
>>> proc = client.query_procedures(id=r.ids)[0]
>>> print(proc)
<ResultRecord(id='0' status='COMPLETE')>
>>> print(proc.properties.scf_total_energy)
-0.6865598095254312
```

Shared Clusters, Supercomputers, and Multiple Clusters

This quickstart guide addresses QCFractal setup on one or more shared cluster(s). The *Server* should be set up on a persistent server for which you have permission to expose ports. For example, this may be a dedicated webserver, the head node of a private cluster, or a cloud instance. The *Manager* should be set up on each shared cluster. In most cases, the *Manager* may be run on the head node; contact your system administrator if you are unsure. This guide requires Parsl to be installed for the *Manager*. It may be installed with `pip` or `conda`.

Begin by initializing the *Server* on your persistent server:

```
qcfractal-server init
```

The QCFractal server receives connections from *Managers* and clients on TCP port 7777. You may optionally specify the `--port` option to choose a custom port. You may need to configure your firewall to allow access to this port.

Because the *Server* will be exposed to the internet, security should be enabled to control access. Enable security by changing the YAML file (default: `~/.qca/qcfractal/qcfractal_config.yaml`) `fractal.security` option to `local`:

```
- security: null
+ security: local
```

Start the *Server*:

```
nohup qcfractal-server start &
```

Note: You may optionally provide a TLS certificate to enable host verification for the *Server* using the `--tls-cert` and `--tls-key` options. If a TLS certificate is not provided, communications with the server will still be encrypted, but host verification will be unavailable (and *Managers* and clients will need to specify `verify=False`).

Next, add users for admin, the *Manager*, and a user (you may choose whatever usernames you like):

```
qcfractal-server user add admin --permissions admin
qcfractal-server user add manager --permissions queue
qcfractal-server user add user --permissions read write compute
```

Passwords will be automatically generated and printed. You may instead specify a password with the `--password` option. See *Fractal Server User* for more information.

Managers should be set up on each shared cluster. In most cases, the *Manager* may be run on the head node; contact your system administrator if you are unsure.

The *Manager* must be configured before use. Create a configuration file (e.g. in `~/.qca/qcfractal/my_manager.yaml`) based on the following template:

```
common:
 adapter: parsl
 tasks_per_worker: 1
 cores_per_worker: 6
 memory_per_worker: 64
 max_workers: 5
 scratch_directory: "$TMPDIR"

cluster:
 node_exclusivity: True
 scheduler: slurm
```

(continues on next page)

(continued from previous page)

```
parsl:
 provider:
  partition: CLUSTER
  cmd_timeout: 30
```

You may need to modify these values to match the particulars of each cluster. In particular:

- The `scheduler` and `partition` options should be set to match the details of your *Scheduler* (e.g. SLURM, PBS, Torque).
- Options related to *Workers* should be set appropriately for the compute node on your cluster. Note that Parsl requires that full nodes be allocated to each *Worker* (i.e. `node_exclusivity: True`).

For more information on *Manager* configuration, see *Fractal Queue Managers* and *Queue Manager Example YAML Files*.

Finally, start the *Manager* in the background on each cluster head node:

```
nohup qcfractal-manager --config-file <path to config YAML> --fractal-uri <URL:port␣
↪of Server> --username manager -password <password> &
```

If you did not specify a TLS certificate in the `qcfractal-server start` step, you will additionally need to specify `--verify False` in the above command.

Test if everything is setup by running a Hartree-Fock calculation on a single hydrogen molecule, as in the *Example* (note this requires `psi4` to be installed on at least one compute resource). This test may be run from any machine.

```
python

>>> import qcfractal.interface as ptl

# Note that server TLS verification may need to be turned off if (verify=False).
# Note that the Server URL and the password for user will need to be filled in.
>>> client = ptl.FractalClient(address="URL:Port", username="user", password="***")
>>> mol = ptl.Molecule(symbols=["H", "H"], geometry=[0, 0, 0, 0, 5, 0])
>>> mol_id = client.add_molecules([mol])[0]
>>> r = client.add_compute("psi4", "HF", "STO-3G", "energy", None, [mol_id])

# Wait a minute for the job to complete
>>> proc = client.query_procedures(id=r.ids)[0]
>>> print(proc)
<ResultRecord(id='0' status='COMPLETE')>
>>> print(proc.properties.scf_total_energy)
-0.6865598095254312
```

Cloud Compute

This quickstart guide addresses QCFractal setup using cloud resources for computation. The *Server* should be set up on a persistent server for which you have permission to expose ports. For example, this may be a dedicated webserver, the head node of a private cluster, or a cloud instance. The *Manager* will be set up on a Kubernetes cluster as a Deployment.

Begin by initializing the *Server* on your persistent server:

```
qcfractal-server init
```

The QCFractal server receives connections from *Managers* and clients on TCP port 7777. You may optionally specify the `--port` option to choose a custom port. You may need to configure your firewall to allow access to this port.

Because the *Server* will be exposed to the internet, security should be enabled to control access. Enable security by changing the YAML file (default: `~/.qca/qcfractal/qcfractal_config.yaml`) `fractal.security` option to `local`:

```
- security: null
+ security: local
```

Start the *Server*:

```
nohup qcfractal-server start &
```

Note: You may optionally provide a TLS certificate to enable host verification for the *Server* using the `--tls-cert` and `--tls-key` options. If a TLS certificate is not provided, communications with the server will still be encrypted, but host verification will be unavailable (and *Managers* and clients will need to specify `verify=False`).

Next, add users for admin, the *Manager*, and a user (you may choose whatever usernames you like):

```
qcfractal-server user add admin --permissions admin
qcfractal-server user add manager --permissions queue
qcfractal-server user add user --permissions read write compute
```

Passwords will be automatically generated and printed. You may instead specify a password with the `--password` option. See *Fractal Server User* for more information.

The *Manager* will be set up on a Kubernetes cluster as a Deployment, running Docker images which each contain QCEngine, QCFractal, and relevant programs. In this guide, we use the molssi/qcarchive_worker_openff Docker image. For execution, this image includes:

- Psi4, dftd3, and MP2D
- RDKit
- geomeTRIC

Note: You may wish to set up a custom Docker image for your specific use case. The Dockerfile corresponding to the molssi/qcarchive_worker_openff image is included below as an example.

```
FROM continuumio/miniconda3
RUN conda install -c psi4/label/dev -c conda-forge psi4 dftd3 mp2d qcengine qcfractal␣
↪rdkit geometric
RUN groupadd -g 999 qcfractal && \
    useradd -m -r -u 999 -g qcfractal qcfractal
USER qcfractal
ENV PATH /opt/local/conda/envs/base/bin/:$PATH
ENTRYPOINT qcfractal-manager --config-file /etc/qcfractal-manager/manager.yaml
```

Create a manager configuration file (e.g. `manager.yaml`) following the template below.

```
common:
 adapter: pool
 tasks_per_worker: 1
 cores_per_worker: 4   # CHANGEME number of cores/worker
```

(continues on next page)

(continued from previous page)

```
 memory_per_worker: 16  # CHANGEME memory/worker in Gb
 max_workers: 1
 scratch_directory: "$TMPDIR"

server:
 fractal_uri: api.qcarchive.molssi.org:443  # CHANGEME URI of your server goes here
 username: manager
 password: foo  # CHANGEME manager password goes here
 verify: True  # False if TLS was skipped earlier

manager:
 manager_name: MyManager  # CHANGEME name your manager
 queue_tag: null
 log_file_prefix: null
 update_frequency: 30
 test: False
```

Add the manager configuration as a secret in Kubernetes:

```
kubectl create secret generic manager-config-yaml --from-file=manager.yaml
```

This allows us to pass the manager configuration into the Docker container securely.

Next, create a Kubernetes deployment configuration file (e.g. `deployment.yaml`) following the template below. The `cpu` and `memory` fields of the deployment configuration should match the `cores_per_worker` and `memory_per_worker` fields of the manager configuration. In this setup, `replicas` determines the number of workers; the `max_workers` and `tasks_per_worker` fields in the manager configuration should be set to 1.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: qcfractal-manager
  labels:
    k8s-app: qcfractal-manager
spec:
  replicas: 4  # CHANGEME: number of images here
  selector:
    matchLabels:
      k8s-app: qcfractal-manager
  template:
    metadata:
      labels:
        k8s-app: qcfractal-manager
    spec:
      containers:
      - image: molssi/qcarchive_worker_openff  # you may wish to specify your own␣
↪Docker image here
        name: qcfractal-manager-pod
        resources:
          limits:
            cpu: 4  # CHANGEME number of cores/worker
            memory: 16Gi  # CHANGEME memory/worker
        volumeMounts:
          - name: manager-config-secret
            mountPath: "/etc/qcfractal-manager"
            readOnly: true
      volumes:
```

(continues on next page)

(continued from previous page)

```
        - name: manager-config-secret
          secret:
            secretName: manager-config-yaml
```

Start the deployment:

```
kubectl apply -f deployment.yaml
```

Note: You can view the status of your deployment with:

```
kubectl get deployments
```

You can view the status of individual "Pods" (Docker containers) with:

```
kubectl get pods --show-labels
```

To get the output of invidual Managers:

```
kubectl logs <pod name>
```

To get Kubernetes metadata and status information about a Pod:

```
kubectl describe pod <pod name>
```

See the Kubernetes Deployment documentation for more information.

Test if everything is setup by running a Hartree-Fock calculation on a single hydrogen molecule, as in the *Example* (note this requires `psi4` to be installed on at least one compute resource). This test may be run from any machine.

```
python

>>> import qcfractal.interface as ptl

# Note that server TLS verification may need to be turned off if (verify=False).
# Note that the Server URL and the password for user will need to be filled in.
>>> client = ptl.FractalClient(address="URL:Port", username="user", password="***")
>>> mol = ptl.Molecule(symbols=["H", "H"], geometry=[0, 0, 0, 0, 5, 0])
>>> mol_id = client.add_molecules([mol])[0]
>>> r = client.add_compute("psi4", "HF", "STO-3G", "energy", None, [mol_id])

# Wait a minute for the job to complete
>>> proc = client.query_procedures(id=r.ids)[0]
>>> print(proc)
<ResultRecord(id='0' status='COMPLETE')>
>>> print(proc.properties.scf_total_energy)
-0.6865598095254312
```

5.3.3 Other Use Cases

QCFractal is highly configurable and supports many use cases beyond those described here. For more information, see the *Server* and *Manager* documentation sections. You may also contact us.

5.4 Server Setup

A `qcfractal-server` instance contains a record of all results, task queue, and collection information and provides an interface to all FractalClients and `qcfractal-managers`. All data is stored in a PostgreSQL database which is often handled transparently. A server instance should be run on hardware that is for long periods stable (not shutdown often), accessible from both compute resources and users via HTTP, and have access to permanent storage. This location is often either research groups local computers, a supercomputer with appropriately allocated resources for this task, or the cloud.

5.4.1 Using the Command Line

The command line is used for `qcfractal-server` instances that are long-term data storage and task distribution engines. To begin, a `qcfractal-server` is first initialized using the command line:

```
>>> qcfractal-server init
```

This initialization will create `~/.qca/qcfractal` folder (which can be altered) which contains default specifications for the `qcfractal-server` and for the underlying PostgreSQL database. The `qcfractal-server init --help` CLI command will describe all parameterizations of this folder. In addition to the specification information, a new PostgreSQL database will be initialized and started in the background. The background PostgreSQL database consumes virtually no resources when not in use and should not interfere with your system.

Once a `qcfractal-server` instance is initialized the server can then be run with the `start` command:

```
>>> qcfractal-server start
```

The QCFractal server is now ready to accept new connections.

5.4.2 Within a Python Script

Canonical workflows can be run from a Python script using the `FractalSnowflake` instance. With default options a `FractalSnowflake` will spin up a fresh database which will be removed after shutdown.

Warning: All data inside a `FractalSnowflake` is temporary and will be deleted when the `FractalSnowflake` shuts down.

```
>>> from qcfractal import FractalSnowflake
>>> server = FractalSnowflake()

# Obtain a FractalClient to the server
>>> client = server.client()
```

A standard `FractalServer` cannot be started in a Python script and then interacted with as a `FractalServer` uses asynchronous programming by default. `FractalServer.stop` will stop the script.

5.4.3 Within a Jupyter Notebook

Due to the way Jupyter Notebooks work an interactive server needs to take a different approach than the canonical Python script. To manipulate a server in a Jupyter Notebook a `FractalSnowflakeHandler` can be used much in the same way as a `FractalSnowflake`.

> **Warning:** All data inside a `FractalSnowflakeHandler` is temporary and will be deleted when the `FractalSnowflakeHandler` shuts down.

```
>>> from qcfractal import FractalSnowflakeHandler
>>> server = FractalSnowflakeHandler()

# Obtain a FractalClient to the server
>>> client = server.client()
```

5.4.4 Full Server Config Settings

The full CLI and configs for the Fractal Server can be found on the following pages:

- Fractal Server Config file: *Fractal Server Config*
- `qcfractal-server init`: *Fractal Server Init*
- `qcfractal-server start`: *Fractal Server Start*
- `qcfractal-server upgrade`: *Fractal Server Upgrade*

5.5 Manager Setup

Once a QCFractal server is running, compute can be attached to it by spinning up `qcfractal-manager`. These `qcfractal-manager` connect to your FractalServer instance, adds tasks to a distributed workflow manager, and pushes complete tasks back to the `qcfractal-server` instance. These `qcfractal-manager` should be run on either the machine that is executing the computations or on the head nodes of supercomputers and local clusters.

5.5.1 Distributed Workflow Engines

QCFractal supports a number of distributed workflow engines to execute computations. Each of these has strengths and weaknesses depending on the workload, task specifications, and resources that the compute will be executed on. In general, we recommend the following:

- For laptops and single nodes: ProcessPoolExecutor
- For local clusters: Dask or Parsl

The ProcessPoolExecutor uses built-in Python types and requires no additional libraries while Dask requires `dask`, `dask.distributed`, and `dask_jobqueue` (Dask Distributed Docs, Dask Jobqueue Docs); and Parsl requires `parsl` (Parsl Docs)

5.5.2 Using the Command Line

Note: The CLI + YAML config file is the current recommended way to start and run Fractal Queue Managers

At the moment only ProcessPoolExecutor `qcfractal-manager` can be spun up purely from the command line as other distributed workflow managers require additional setup through a YAML config file.

For the full docs for setting up a *Manager*, please see *the Manager documentation pages.*

Launching a `qcfractal-manager` using a ProcessPoolExecutor:

```
$ fractal-manager executor
[I 190301 10:45:50 managers:118] QueueManager:
[I 190301 10:45:50 managers:119]     Version:         v0.5.0

[I 190301 10:45:50 managers:122]     Name Information:
[I 190301 10:45:50 managers:123]         Cluster:     unknown
[I 190301 10:45:50 managers:124]         Hostname:    qcfractal.local
[I 190301 10:45:50 managers:125]         UUID:        0d2b7704-6ac0-4ef7-b831-
↪00aa6afa8c1c

[I 190301 10:45:50 managers:127]     Queue Adapter:
[I 190301 10:45:50 managers:128]         <ExecutorAdapter client=<ProcessPoolExecutor␣
↪max_workers=8>>

[I 190301 10:45:50 managers:131]     QCEngine:
[I 190301 10:45:50 managers:132]         Version:    v0.6.1

[I 190301 10:45:50 managers:150]     Connected:
[I 190301 10:45:50 managers:151]         Version:     v0.5.0
[I 190301 10:45:50 managers:152]         Address:     https://localhost:7777/
[I 190301 10:45:50 managers:153]         Name:        QCFractal Server
[I 190301 10:45:50 managers:154]         Queue tag:   None
[I 190301 10:45:50 managers:155]         Username:    None

[I 190301 10:45:50 managers:194] QueueManager successfully started. Starting IOLoop.
```

The connected `qcfractal-server` instance can be controlled by:

```
$ qcfractal-manager --fractal-uri=api.qcfractal.molssi.org:443
```

Only basic settings can be started through the CLI and most of the options require a YAML config file to get up and going. You can check all valid YAML options in *the Manager documentation pages* or you can always check the current schema from the CLI with:

```
$ qcfractal-manager --schema
```

The CLI has several options which can examined with:

```
qcfractal-manager --help
```

Every option specified in the CLI has an equal option in the YAML config file (except for `--help` and `--schema`), but many YAML options are not present in the CLI due to their complex nature. Any option set in both places will defer to the CLI's setting, allowing you to overwrite some of the common YAML config options on invocation.

Note: The `--manager-name` argument is useful to change the name of the manager reported back to the *Server*

instance. In addition, the `--queue-tag` will limit the acquisition of tasks to only the desired *Server* task tags. These settings can also all be set in the YAML config file.

5.5.3 Using the Python API

`qcfractal-managers` can also be created using the Python API.

> **Warning:** This is for advanced users and special care needs to be taken to ensure that both the manager and the workflow tool need to understand the number of cores and memory available to prevent over-subscription of compute.

```
from qcfractal.interface import FractalClient
from qcfractal import QueueManager

import dask import distributed

fractal_client = FractalClient("localhost:7777")
workflow_client = distributed.Client("tcp://10.0.1.40:8786")

ncores = 4
mem = 2

# Build a manager
manager = QueueManager(fractal_client, workflow_client, cores_per_task=ncores, memory_
↪per_task=mem)

# Important for a calm shutdown
from qcfractal.cli.cli_utils import install_signal_handlers
install_signal_handlers(manager.loop, manager.stop)

# Start or test the loop. Swap with the .test() and .start() method respectively
manager.start()
```

5.5.4 Testing

A `qcfractal-manager` can be tested using the `--test` argument and does not require an active `qcfractal-manager`, this is very useful to check if both the distributed workflow manager is setup correctly and correct computational engines are found.

```
$ qcfractal-manager --test
[I 190301 10:55:57 managers:118] QueueManager:
[I 190301 10:55:57 managers:119]     Version:         v0.5.0+52.g6eab46f

[I 190301 10:55:57 managers:122]     Name Information:
[I 190301 10:55:57 managers:123]         Cluster:     unknown
[I 190301 10:55:57 managers:124]         Hostname:    Daniels-MacBook-Pro.local
[I 190301 10:55:57 managers:125]         UUID:        0cd257a6-c839-4743-bb33-
↪fa55bebac1e1

[I 190301 10:55:57 managers:127]     Queue Adapter:
[I 190301 10:55:57 managers:128]         <ExecutorAdapter client=<ProcessPoolExecutor␣
↪max_workers=8>>
```

(continues on next page)

(continued from previous page)

```
[I 190301 10:55:57 managers:131]     QCEngine:
[I 190301 10:55:57 managers:132]         Version:    v0.6.1

[I 190301 10:55:57 managers:158]     QCFractal server information:
[I 190301 10:55:57 managers:159]         Not connected, some actions will not be␣
↪available
[I 190301 10:55:57 managers:389] Testing requested, generating tasks
[I 190301 10:55:57 managers:425] Found program rdkit, adding to testing queue.
[I 190301 10:55:57 managers:425] Found program torchani, adding to testing queue.
[I 190301 10:55:57 managers:425] Found program psi4, adding to testing queue.
[I 190301 10:55:57 base_adapter:124] Adapter: Task submitted rdkit
[I 190301 10:55:57 base_adapter:124] Adapter: Task submitted torchani
[I 190301 10:55:57 base_adapter:124] Adapter: Task submitted psi4
[I 190301 10:55:57 managers:440] Testing tasks submitting, awaiting results.

[I 190301 10:56:04 managers:444] Testing results acquired.
[I 190301 10:56:04 managers:451] All tasks retrieved successfully.
[I 190301 10:56:04 managers:456]   rdkit - PASSED
[I 190301 10:56:04 managers:456]   torchani - PASSED
[I 190301 10:56:04 managers:456]   psi4 - PASSED
[I 190301 10:56:04 managers:465] All tasks completed successfully!
```

Records Documentation

The records created from adding additional compute.

- *Results*
- *Procedures*
- *Services*
- *Fractal Call Flowcharts*

5.6 Results

A result is a single quantum chemistry method evaluation, which might be an energy, an analytic gradient or Hessian, or a property evaluation. Collections of evaluations such as finite-difference gradients, complete basis set extrapolation, or geometry optimizations belong under the "Procedures" heading.

5.6.1 Indices

A result can be found based off a unique tuple of `(driver, program, molecule_id, keywords_set, method, basis)`

- `driver` - The type of calculation being evaluated (i.e. `energy`, `gradient`, `hessian`, `properties`)
- `program` - A lowercase string representation of the quantum chemistry program used (`gamess`, `nwchem`, `psi4`, etc.)
- `molecule_id` - The *ObjectId* of the molecule used in the computation.
- `keywords_set` - The key to the options set stored in the database (e.g. `default` -> `{"e_convergence": 1.e-7, "scf_type": "df", ...}`)

- `method` - A lowercase string representation of the method used in the computation (e.g. `b3lyp`, `mp2`, `ccsd(t)`).
- `basis` - A lowercase string representation of the basis used in the computation (e.g. `6-31g`, `cc-pvdz`, `def2-svp`)

5.6.2 Schema

All results are stored using the QCSchema so that the storage is quantum chemistry program agnostic. An example of the QCSchema input is shown below:

```
{
  "schema_name": "qc_json_input",
  "schema_version": 1,
  "molecule": {
    "geometry": [
      0.0,  0.0,    -0.1294,
      0.0, -1.4941,  1.0274,
      0.0,  1.4941,  1.0274
    ],
    "symbols": ["O", "H", "H"]
  },
  "driver": "energy",
  "model": {
    "method": "MP2",
    "basis": "cc-pVDZ"
  },
  "keywords": {},
}
```

This input would correspond to the following output:

```
{
  "schema_name": "qc_json_output",
  "schema_version": 1,
  "molecule": {
    "geometry": [
      0.0,  0.0,    -0.1294,
      0.0, -1.4941,  1.0274,
      0.0,  1.4941,  1.0274
    ],
    "symbols": ["O", "H", "H"]
  },
  "driver": "energy",
  "model": {
    "method": "MP2",
    "basis": "cc-pVDZ"
  },
  "keywords": {},
  "provenance": {
    "creator": "QM Program",
    "version": "1.1",
    "routine": "module.json.run_json"
  },
  "return_result": -76.22836742810021,
  "success": true,
  "properties": {
```

(continues on next page)

(continued from previous page)


```
    "calcinfo_nbasis": 24,
    "calcinfo_nmo": 24,
    "calcinfo_nalpha": 5,
    "calcinfo_nbeta": 5,
    "calcinfo_natom": 3,
    "return_energy": -76.22836742810021,
    "scf_one_electron_energy": -122.44534536383037,
    "scf_two_electron_energy": 37.62246494040059,
    "nuclear_repulsion_energy": 8.80146205625184,
    "scf_dipole_moment": [0.0, 0.0, 2.0954],
    "scf_iterations": 10,
    "scf_total_energy": -76.02141836717794,
    "mp2_same_spin_correlation_energy": -0.051980792916251864,
    "mp2_opposite_spin_correlation_energy": -0.15496826800602342,
    "mp2_singles_energy": 0.0,
    "mp2_doubles_energy": -0.20694906092226972,
    "mp2_total_correlation_energy": -0.20694906092226972,
    "mp2_total_energy": -76.22836742810021
  }
}
```


5.7 Procedures

5.7.1 Pictorial Procedure Flowchart

See the *flowchart showing the psuedo-calls* made when the Client adds a procedure to Fractal.

5.8 Services

Services are unique workflows where there is an iterative component on the server. A typical service workflow looks like the following:

1. A client submits a new service request to the server.
2. A service is created on the server and placed in the service queue.
3. A service iteration is called that will spawn new tasks.
4. The service waits until all generated tasks are complete.
5. The service repeats 3 and 4 until the service iterations are complete.
6. The service cleans intermediate data, finalizes the data representation, and marks itself complete.

The TorsionDrive service will be used as an example to illuminate the above steps. The TorsionDrive service optimizes the geometry of a biomolecule at a number of frozen dihedral angles to provide an energy profile of the rotation of this dihedral bond.

Consider the service using a concrete example of scanning the hydrogen peroxide dihedral:

1. A client submits a task to scan the HOOH molecule dihedral every 90 degrees as a service.
2. The service is received by the server, and the first 0-degree dihedral geometry optimization *Task* is spawned.
3. The service waits until the 0-degree *Task* is complete, and then generates 90 and -90-degree *tasks* based off this 0-degree geometry.

4. The service waits for the two new *tasks* to complete and spawns 0 and 180-degree tasks based on the 90 and -90-degree geometries.
5. The service waits for the 90- and -90-degree *tasks* to complete. Then it builds its final data structure for user querying and marks itself complete.

The service technology allows the `FractalServer` to complete very complex workflows of arbitrary design. To see a pictorial representation of this process, please see the *flowchart showing the pseudo-calls* when a service is added to the `FractalServer`

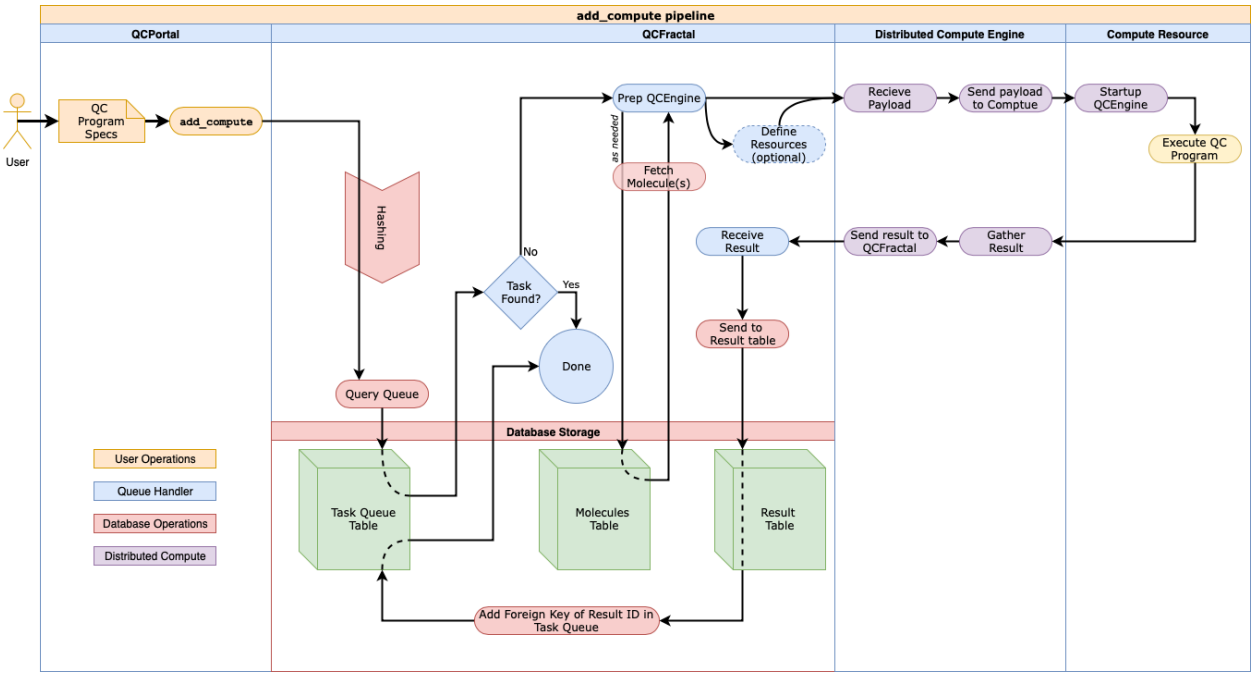
5.8.1 TorsionDrive

5.9 Fractal Call Flowcharts

The interface between the Portal client, the Fractal server, and the distributed compute resources is not something easily conveyed by text. We have created flowchart diagrams to help explain what happens from the time Portal invokes a call to Fractal, to the time that Fractal finishes handling the request. These diagrams are simplified to not show every routine and middleware call, but instead to provide a visual aid to what is happening to help understanding.

5.9.1 `add_compute`

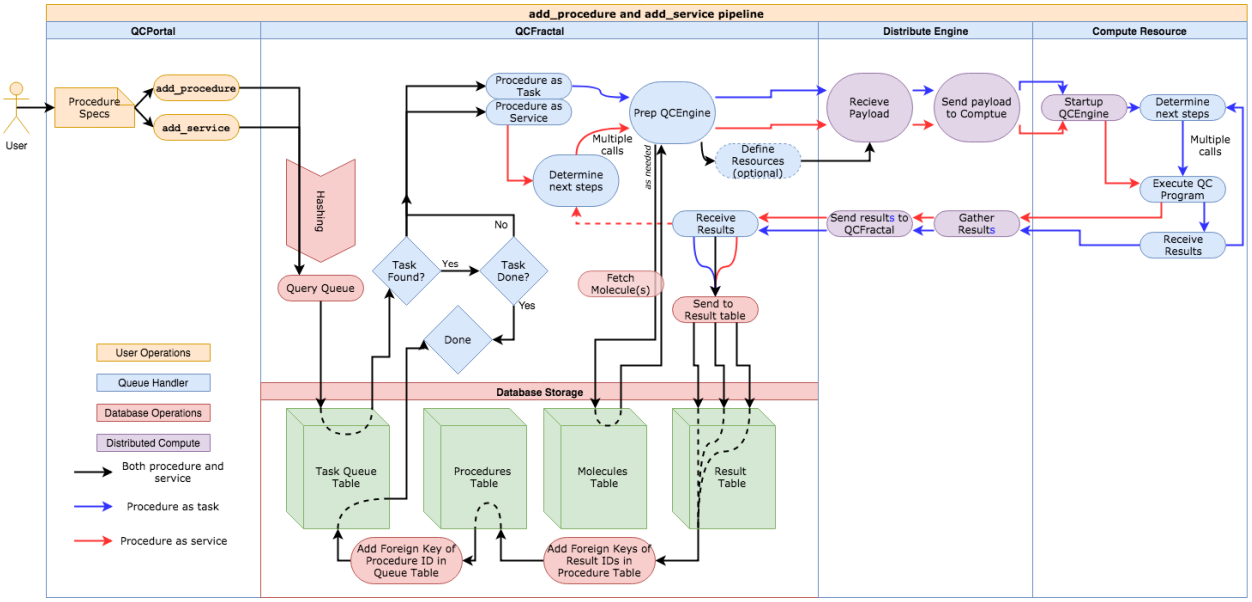
This flowchart follows the logic behind a user's call to add a compute action to fractal and any attached distributed compute system.



5.9.2 `add_procedure` or `add_service`

When a user calls `add_procedure` or `add_service`, much of the same logic is called. The major difference is which side of the distributed compute engine the logic of the subsequent procedural calls are handled, on the compute side, or the Fractal side.

This flowchart shows both ends and provides a different path for each call show by the different colored connecting arrows.



Manager Documentation

Setting up and running Fractal's Queue Managers on your system.

- *Fractal Queue Managers*
- *Queue Manager API*
- *Configuration for High-Performance Computing*
- *Queue Manager Example YAML Files*
- *Queue Manager Frequent Questions and Issues*

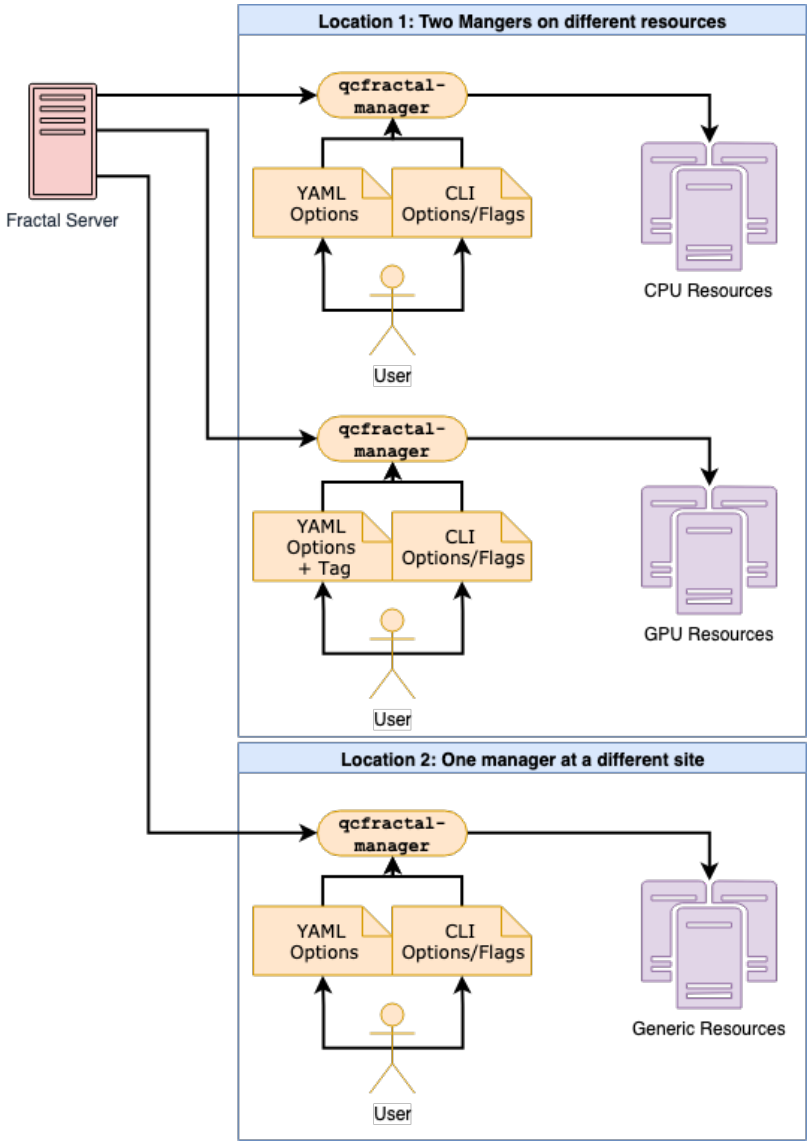
5.10 Fractal Queue Managers

Queue Managers are the processes which interface with the Fractal *Server* and clusters, supercomputers, and cloud resources to execute the tasks in the Fractal *Server*. These managers pull compute *tasks* from the *Server*, and then pass them to various distributed back ends for computation for a variety of different needs. The architecture of the Fractal *Server* allows many managers to be created in multiple physical locations. Currently, Fractal supports the following:

- *Pool* - A python *ProcessPoolExecutor* for computing tasks on a single machine (or node).
- Dask - A graph-based workflow engine for laptops and small clusters.
- Parsl - High-performance workflows.
- Fireworks - An asynchronous Mongo-based distributed queuing system.

These backends allow Fractal to be incredibly elastic in utilizing computational resources, scaling from a single laptop to thousands of nodes on physically separate hardware. Our end goal is to be able to setup a manager at a physical site and allow it to scale up and down as its task queue requires and execute compute over long periods of time (months) without intervention.

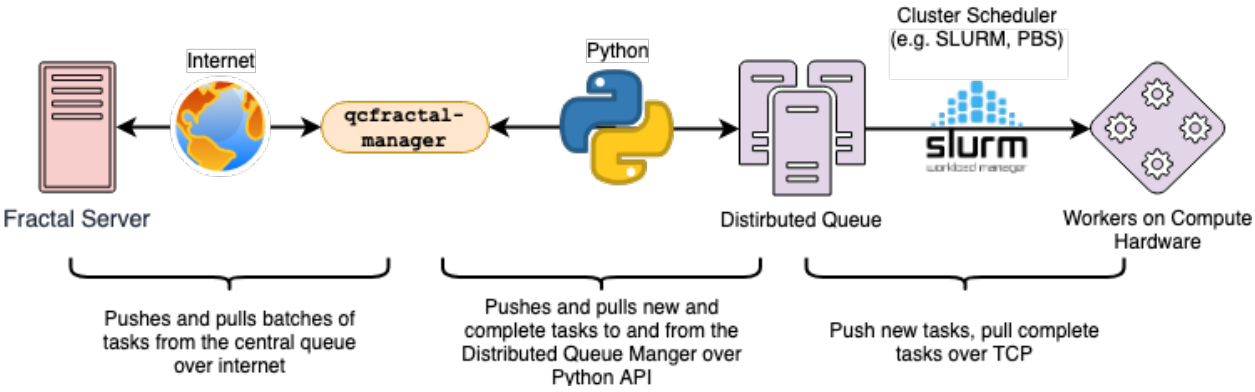
The basic setup of the Queue *Managers* and how they interact with the *Server* is as follows:



In this, multiple *managers* talk to a central *Fractal Server* and deploy *tasks* to different compute resources based on availability, physical location, and *tags*.

The main goals of the Queue *Manager* is to reduce the user's level of expertise needed to start compute with Fractal and, more importantly, to need as little manual intervention as possible to have persistent compute. Ideally, you start the *Manager* in a background process, leave it be while it checks in with the *Fractal Server* from time to time to get *tasks*, and pushes/pulls *tasks* from the distributed *Adapter* as need be.

The communication between each of the layers involved, and the mechanism by which they communicate is summarized in this image:



The different levels of communication are all established automatically once the user configures the manager, and this image shows how information flow from point-to-point.

The manager itself is a fairly lightweight process and consumes very little CPU power on its own. You should talk with your sysadmin before running this on a head node, but the Queue *Manager* itself will consume less than 1% CPU we have found and virtually no RAM.

If you are interested in the more detailed workings of the *Manager*, please see the *Detailed Manager Information* page. However, the information on that page is not required to set up and run a *Manager*.

5.10.1 Queue Manager Quick Starts

For those who just want to get up and going, consider the following examples.

Laptop/Desktop Quick Boot

To get a Manager set up with defaults, running on local hardware, consuming local CPU and RAM, targeting a Fractal Server running locally, run the following:

```
$ qcfractal-manager
```

SLURM Cluster, Dask Adapter

To start a manager with a dask *Adapter*, on a SLURM cluster, consuming 1 CPU and 8 GB of ram, targeting a Fractal Server running on that cluster, and using the SLURM partition `default`, save the following YAML config file:

```
common:
 adapter: dask
 tasks_per_worker: 1
 cores_per_worker: 1
 memory_per_worker: 8

cluster:
 scheduler: slurm

dask:
 queue: default
```

and then run the following command:

```
$ qcfractal-manager --config-file="path/to/config.yaml"
```

replacing the `config-file` arg with the path to the file you saved. You will need `dask` and `dask-jobqueue` (Dask Distributed Docs, Dask Jobqueue Docs), to run this example, which are not packages required by Fractal unless you are running compute managers; if you use a different *Adapter*, you would need a separate set of packages.

5.10.2 Queue Manager CLI

The CLI for the Fractal Queue Manager acts as an **option-specific** overwrite of the YAML file for various options and therefore its flags can be set in tandem with the YAML. However, it does not have as extensive control as the YAML file and so complex Managers (like those running Dask and Parsl) need to be setup in YAML.

In case this ever falls out of date, you can always run `qcfractal-manager --help` to get the most up-to-date help block.

```
$ qcfractal-manager --help

usage: qcfractal-manager [-h] [--config-file CONFIG_FILE] [--adapter ADAPTER]
                         [--tasks_per_worker TASKS_PER_WORKER]
                         [--cores-per-worker CORES_PER_WORKER]
                         [--memory-per-worker MEMORY_PER_WORKER]
                         [--scratch-directory SCRATCH_DIRECTORY] [-v]
                         [--fractal-uri FRACTAL_URI] [-u USERNAME]
                         [-p PASSWORD] [--verify VERIFY]
                         [--manager-name MANAGER_NAME] [--queue-tag QUEUE_TAG]
                         [--log-file-prefix LOG_FILE_PREFIX]
                         [--update-frequency UPDATE_FREQUENCY]
                         [--max-queued-tasks MAX_QUEUED_TASKS] [--test]
                         [--ntests NTESTS] [--schema]

A CLI for a QCFractal QueueManager with a ProcessPoolExecutor, Dask, or Parsl
backend. The Dask and Parsl backends *requires* a config file due to the
complexity of its setup. If a config file is specified, the remaining options
serve as CLI overwrites of the config.

optional arguments:
  -h, --help            show this help message and exit
  --config-file CONFIG_FILE

Common Adapter Settings:
  --adapter ADAPTER     The backend adapter to use, currently only {'dask',
                        'parsl', 'pool'} are valid.
  --tasks_per_worker TASKS_PER_WORKER
                        The number of simultaneous tasks for the executor to
                        run, resources will be divided evenly.
  --cores-per-worker CORES_PER_WORKER
                        The number of process for each executor's Workers
  --memory-per-worker MEMORY_PER_WORKER
                        The total amount of memory on the system in GB
  --scratch-directory SCRATCH_DIRECTORY
                        Scratch directory location
  -v, --verbose         Increase verbosity of the logger.

FractalServer connection settings:
  --fractal-uri FRACTAL_URI
                        FractalServer location to pull from
```

(continues on next page)

(continued from previous page)

```
  -u USERNAME, --username USERNAME
                        FractalServer username
  -p PASSWORD, --password PASSWORD
                        FractalServer password
  --verify VERIFY       Do verify the SSL certificate, leave off (unset) for
                        servers with custom SSL certificates.

QueueManager settings:
  --manager-name MANAGER_NAME
                        The name of the manager to start
  --queue-tag QUEUE_TAG
                        The queue tag to pull from
  --log-file-prefix LOG_FILE_PREFIX
                        The path prefix of the logfile to write to.
  --update-frequency UPDATE_FREQUENCY
                        The frequency in seconds to check for complete tasks.
  --max-queued-tasks MAX_QUEUED_TASKS
                        Maximum number of tasks to hold at any given time.
                        Generally should not be set.

Optional Settings:
  --test                Boot and run a short test suite to validate setup
  --ntests NTESTS       How many tests per found program to run, does nothing
                        without --test set
  --schema              Display the current Schema (Pydantic) for the YAML
                        config file and exit. This will always show the most
                        up-to-date schema. It will be presented in a JSON-like
                        format.
```

5.10.3 Terminology

There are a number of terms which can overlap due to the layers of abstraction and the type of software and hardware the Queue Manager interacts with. To help with that, the pages in this section will use the terminology defined below. Several pieces of software we interface with may have their own terms or the same term with different meaning, but because one goal of the Manager is to abstract those concepts away as best it can, we choose the following set. If you find something inappropriately labeled, unclear, or overloaded in any way, please raise an issue on GitHub and help us make it better!

An important note: Not all the concepts/mechanics of the *Manager* and *Adapter* are covered in the glossary by design! There are several abstraction layers and mechanics which the user should never have to interact with or even be aware of. However, if you feel something is missing, let us know!

Adapter The specific piece of software which accepts *tasks* from the *Manager* and sends them to the physical hardware. It is also the software which typically interacts with a cluster's *Scheduler* to allocate said hardware and start *Job*.

Distributed Compute Engine A more precise, although longer-winded, term for the *Adapter*.

Job The specific allocation of resources (CPU, Memory, wall clock, etc) provided by the *Scheduler* to the *Adapter*. This is identical to if you requested batch-like job on a cluster (e.g. though `qsub` or `sbatch`), however, it is more apt to think of the resources allocated in this way as "resources to be distributed to the *Task* by the *Adapter*". Although a user running a *Manager* will likely not directly interact with these, its important to track as these are what your *Scheduler* is actually running and your allocations will be charged by. At least (and usually only) one *Worker* will be deployed to a *Job* from the *Adapter* to handle incoming *Task*s. Once the *Worker* lands, it will report back to the *Adapter* and all communications happen between those two objects; the Job simply runs until either the *Adapter* stops it, or the *Scheduler* ends it.

Manager The *Fractal Queue Manager*. The term "Manager" presented by itself refers to this object.

Scheduler The software running on a cluster which users request hardware from to run computational *tasks*, e.g. PBS, SLURM, LSF, SGE, etc. This, by itself, does not have any concept of the *Manager* or even the *Adapter* as both interface with *it*, not the other way around. Individual users' clusters may, and almost always, have a different configuration, even amongst the same governing software. Therefore, no two Schedulers should be treated the same. In many cases, the *Adapter* submits a *Job* to the Scheduler with instructions of how the *Job* should start a *Worker* once it is allocated and booted.

Server The Fractal Server that the *Manager* connects to. This is the source of the *Task*s which are pulled from and pushed to. Only the *Manager* has any notion of the Server of all the other software involved with the *Manager* does not.

Tag Arbitrary categorization labels that different *tasks* can be assigned when submitted to the *Server*. *Managers* can pull these tags if configured, and will *exclusively* pull their defined tag if so. Similarly, *tasks* set with a given tag can *only* be pulled if their *Manager* is configured to do so.

Task A single unit of compute as defined by the Fractal *Server* (i.e. the item which comes from the Task Queue). These tasks are preserved as they pass to the distributed compute engine and are what are presented to each distributed compute engine's *Worker*s to compute

Worker The process executed from the *Adapter* on the allocated hardware inside a *Job*. This process receives the *tasks* tracked by the *Adapter* and is responsible for their execution. The Worker itself is responsible for consuming the resources of the *Job* and distributing them to handle concurrent *tasks*. In most cases, there will be 1 Worker per *Job*, but there are some uncommon instances where this isn't true. You can safely assume the 1 Worker/*Job* case for Fractal usage. Resources allocated for the Worker will be distributed by the *Adapter*s configuration, but is usually uniform.

5.11 Queue Manager API

This page documents **all** valid options for the YAML file inputs to the config manager. This first section outlines each of the headers (top level objects) and a description for each one. The final file will look like the following:

```
common:
    option_1: value_for1
    another_opt: 42
server:
    option_for_server: "some string"
```

This is the complete set of options, auto-generated from the parser itself, so it should be accurate for the given release. If you are using a developmental build or want to see the schema yourself, you can run the `qcfractal-manager --schema` command and it will display the whole schema for the YAML input.

Each section below here is summarized the same way, showing all the options for that YAML header in the form of their pydantic API which the YAML is fed into in a one-to-one match of options.

class qcfractal.cli.qcfractal_manager.**ManagerSettings**(*, *common: qcfractal.cli.qcfractal_manager.CommonManagerSettings = CommonManagerSettings(adapter=<AdapterEnum.pool: 'pool'>, tasks_per_worker=1, cores_per_worker=2, memory_per_worker=6.704, max_workers=1, retries=2, scratch_directory=None, verbose=False, nodes_per_job=1, nodes_per_task=1, cores_per_rank=1)*, *server: qcfractal.cli.qcfractal_manager.FractalServerSettings = FractalServerSettings(fractal_uri='localhost:7777', username=None, password=None, verify=None)*, *manager: qcfractal.cli.qcfractal_manager.QueueManagerSettings = QueueManagerSettings(manager_name='unlabeled', queue_tag=None, log_file_prefix=None, update_frequency=30.0, test=False, ntests=5, max_queued_tasks=None)*, *cluster: qcfractal.cli.qcfractal_manager.ClusterSettings = ClusterSettings(node_exclusivity=False, scheduler=None, scheduler_options=[], task_startup_commands=[], walltime='06:00:00', adaptive=<AdaptiveCluster.adaptive: 'adaptive'>)*, *dask: qcfractal.cli.qcfractal_manager.DaskQueueSettings = DaskQueueSettings(interface=None, extra=None, lsf_units=None)*, *parsl: qcfractal.cli.qcfractal_manager.ParslQueueSettings = ParslQueueSettings(executor=ParslExecutorSettings(address=None) provider=ParslProviderSettings(partition=None, launcher=None))*)

The config file for setting up a QCFractal Manager, all sub fields of this model are at equal top-level of the YAML file. No additional top-level fields are permitted, but sub-fields may have their own additions.

Not all fields are required and many will depend on the cluster you are running, and the adapter you choose to run on.

Parameters

- **common** (*CommonManagerSettings*, Optional)
- **server** (*FractalServerSettings*, Optional)
- **manager** (*QueueManagerSettings*, Optional)
- **cluster** (*ClusterSettings*, Optional)
- **dask** (*DaskQueueSettings*, Optional)
- **parsl** (*ParslQueueSettings*, Optional)

5.11.1 common

class qcfractal.cli.qcfractal_manager.**CommonManagerSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>', _env_file_encoding: Optional[str] = None, _secrets_dir: Optional[Union[pathlib.Path, str]] = None, *, adapter: qcfractal.cli.qcfractal_manager.AdapterEnum = <AdapterEnum.pool: 'pool'>, tasks_per_worker: int = 1, cores_per_worker: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 2, memory_per_worker: qcfractal.cli.qcfractal_manager.ConstrainedFloatValue = 6.704, max_workers: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 1, retries: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 2, scratch_directory: str = None, verbose: bool = False, nodes_per_job: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 1, nodes_per_task: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 1, cores_per_rank: int = 1*)

The Common settings are the settings most users will need to adjust regularly to control the nature of task execution and the hardware under which tasks are executed on. This block is often unique to each deployment, user, and manager and will be the most commonly updated options, even as config files are copied and reused, and even on the same platform/cluster.

Parameters

- **adapter** (*{dask,pool,parsl}, Default: pool*) – Which type of Distributed adapter to run tasks through.
- **tasks_per_worker** (*int, Default: 1*) – Number of concurrent tasks to run *per Worker* which is executed. Total number of concurrent tasks is this value times max_workers, assuming the

hardware is available. With the pool adapter, and/or if max_workers=1, tasks_per_worker *is* the number of concurrent tasks.

- **cores_per_worker** (*ConstrainedInt, Default: 2*) – Number of cores to be consumed by the Worker and distributed over the tasks_per_worker. These cores are divided evenly, so it is recommended that quotient of cores_per_worker/tasks_per_worker be a whole number else the core distribution is left up to the logic of the adapter. The default value is read from the number of detected cores on the system you are executing on.

  In the case of node-parallel tasks, this number means the number of cores per node.

- **memory_per_worker** (*ConstrainedFloat, Default: 6.704*) – Amount of memory (in GB) to be consumed and distributed over the tasks_per_worker. This memory is divided evenly, but is ultimately at the control of the adapter. Engine will only allow each of its calls to consume memory_per_worker/tasks_per_worker of memory. Total memory consumed by this manager at any one time is this value times max_workers. The default value is read from the amount of memory detected on the system you are executing on.

- **max_workers** (*ConstrainedInt, Default: 1*) – The maximum number of Workers which are allowed to be run at the same time. The total number of concurrent tasks will maximize at this quantity times tasks_per_worker.The total number of Jobs on a cluster which will be started is equal to this parameter in most cases, and should be assumed 1 Worker per Job. Any exceptions to this will be documented. In node exclusive mode this is equivalent to the maximum number of nodes which you will consume. This must be a positive, non zero integer.

- **retries** (*ConstrainedInt, Default: 2*) – Number of retries that QCEngine will attempt for RandomErrors detected when running its computations. After this many attempts (or on any other type of error), the error will be raised.

- **scratch_directory** (*str, Optional*) – Scratch directory for Engine execution jobs.

- **verbose** (*bool, Default: False*) – Turn on verbose mode or not. In verbose mode, all messages from DEBUG level and up are shown, otherwise, defaults are all used for any logger.

- **nodes_per_job** (*ConstrainedInt, Default: 1*) – The number of nodes to request per job. Only used by the Parsl adapter at present

- **nodes_per_task** (*ConstrainedInt, Default: 1*) – The number of nodes to use for each tasks. Only relevant for node-parallel executables.

- **cores_per_rank** (*int, Default: 1*) – The number of cores per MPI rank for MPI-parallel applications. Only relevant for node-parallel codes and the most relevant to codes that with hybrid MPI+OpenMP parallelism (e.g., NWChem).

5.11.2 server

class qcfractal.cli.qcfractal_manager.**FractalServerSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>', _env_file_encoding: Optional[str] = None, _secrets_dir: Optional[Union[pathlib.Path, str]] = None, *, fractal_uri: str = 'localhost:7777', username: str = None, password: str = None, verify: bool = None*)

Settings pertaining to the Fractal Server you wish to pull tasks from and push completed tasks to. Each manager supports exactly 1 Fractal Server to be in communication with, and exactly 1 user on that Fractal Server. These can be changed, but only once the Manager is shutdown and the settings changed. Multiple Managers however can be started in parallel with each other, but must be done as separate calls to the CLI.

Caution: The password here is written in plain text, so it is up to the owner/writer of the configuration file to ensure its security.

Parameters

- **fractal_uri** (*str, Default: localhost:7777*) – Full URI to the Fractal Server you want to connect to
- **username** (*str, Optional*) – Username to connect to the Fractal Server with. When not provided, a connection is attempted as a guest user, which in most default Servers will be unable to return results.
- **password** (*str, Optional*) – Password to authenticate to the Fractal Server with (alongside the *username*)
- **verify** (*bool, Optional*) – Use Server-side generated SSL certification or not.

5.11.3 manager

class qcfractal.cli.qcfractal_manager.**QueueManagerSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>', _env_file_encoding: Optional[str] = None, _secrets_dir: Optional[Union[pathlib.Path, str]] = None*, **, manager_name: str = 'unlabeled', queue_tag: Optional[Union[str, List[str]]] = None, log_file_prefix: str = None, update_frequency: qcfractal.cli.qcfractal_manager.ConstrainedFloatVal = 30, test: bool = False, ntests: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = 5, max_queued_tasks: qcfractal.cli.qcfractal_manager.ConstrainedIntValue = None*)

Fractal Queue Manger settings. These are options which control the setup and execution of the Fractal Manager itself.

Parameters

- **manager_name** (*str, Default: unlabeled*) – Name of this scheduler to present to the Fractal Server. Descriptive names help the server identify the manager resource and assists with debugging.
- **queue_tag** (*Union[str, List[str]], Optional*) – Only pull tasks from the Fractal Server with this tag. If not set (None/null), then pull untagged tasks, which should be the majority of tasks. This option should only be used when you want to pull very specific tasks which you know have been tagged as such on the server. If the server has no tasks with this tag, no tasks will be pulled (and no error is raised because this is intended behavior). If multiple tags are provided, tasks will be pulled (but not necessarily executed) in order of the tags.
- **log_file_prefix** (*str, Optional*) – Full path to save a log file to, including the filename. If not provided, information will still be reported to terminal, but not saved. When set, logger information is sent both to this file and the terminal.
- **update_frequency** (*ConstrainedFloat, Default: 30*) – Time between heartbeats/update checks between this Manager and the Fractal Server. The lower this value, the shorter the intervals. If you have an unreliable network connection, consider increasing this time as repeated, consecutive network failures will cause the Manager to shut itself down to maintain integrity between it and the Fractal Server. Units of seconds
- **test** (*bool, Default: False*) – Turn on testing mode for this Manager. The Manager will not connect to any Fractal Server, and instead submits netsts worth trial tasks per quantum chemistry program it finds. These tasks are generated locally and do not need a running

Fractal Server to work. Helpful for ensuring the Manager is configured correctly and the quantum chemistry codes are operating as expected.

- **ntests** (*ConstrainedInt, Default: 5*) – Number of tests to run if the *test* flag is set to True. Total number of tests will be this number times the number of found quantum chemistry programs. Does nothing if *test* is False.If set to 0, then this submits no tests, but it will run through the setup and client initialization.
- **max_queued_tasks** (*ConstrainedInt, Optional*) – Generally should not be set. Number of tasks to pull from the Fractal Server to keep locally at all times. If *None*, this is automatically computed as *ceil(common.tasks_per_worker*common.max_workers*2.0) + 1*. As tasks are completed, the local pool is filled back up to this value. These tasks will all attempt to be run concurrently, but concurrent tasks are limited by number of cluster jobs and tasks per job. Pulling too many of these can result in under-utilized managers from other sites and result in less FIFO returns. As such it is recommended not to touch this setting in general as you will be given enough tasks to fill your maximum throughput with a buffer (assuming the queue has them).

5.11.4 cluster

class qcfractal.cli.qcfractal_manager.**ClusterSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>', _env_file_encoding: Optional[str] = None, _secrets_dir: Optional[Union[pathlib.Path, str]] = None, *, node_exclusivity: bool = False, scheduler: qcfractal.cli.qcfractal_manager.SchedulerEnum = None, scheduler_options: List[str] = [], task_startup_commands: List[str] = [], walltime: str = '06:00:00', adaptive: qcfractal.cli.qcfractal_manager.AdaptiveCluster = <AdaptiveCluster.adaptive: 'adaptive'>*)

Settings tied to the cluster you are running on. These settings are mostly tied to the nature of the cluster jobs you are submitting, separate from the nature of the compute tasks you will be running within them. As such, the options here are things like wall time (per job), which Scheduler your cluster has (like PBS or SLURM), etc. No additional options are allowed here.

Parameters

- **node_exclusivity** (*bool, Default: False*) – Run your cluster jobs in node-exclusivity mode. This option may not be available to all scheduler types and thus may not do anything. Related to this, the flags we have found for this option may not be correct for your scheduler and thus might throw an error. You can always add the correct flag/parameters to the *scheduler_options* parameter and leave this as False if you find it gives you problems.
- **scheduler** (*{slurm,pbs,sge,moab,lsf,cobalt}, Optional*) – Option of which Scheduler/Queuing system your cluster uses. Note: not all scheduler options are available with every adapter.

- **scheduler_options** (*List[str], Default: []*) – Additional options which are fed into the header files for your submitted jobs to your cluster's Scheduler/Queuing system. The directives are automatically filled in, so if you want to set something like '#PBS -n something', you would instead just do '-n something'. Each directive should be a separate string entry in the list. No validation is done on this with respect to valid directives so it is on the user to know what they need to set.
- **task_startup_commands** (*List[str], Default: []*) – Additional commands to be run before starting the Workers and the task distribution. This can include commands needed to start things like conda environments or setting environment variables before executing the Workers. These commands are executed first before any of the distributed commands run and are added to the batch scripts as individual commands per entry, verbatim.
- **walltime** (*str, Default: 06:00:00*) – Wall clock time of each cluster job started. Presented as a string in HH:MM:SS form, but your cluster may have a different structural syntax. This number should be set high as there should be a number of Fractal tasks which are run for each submitted cluster job. Ideally, the job will start, the Worker will land, and the Worker will crunch through as many tasks as it can; meaning the job which has a Worker in it must continue existing to minimize time spend redeploying Workers.
- **adaptive** (*{static,adaptive}, Default: adaptive*) – Whether or not to use adaptive scaling of Workers or not. If set to 'static', a fixed number of Workers will be started (and likely *NOT* restarted when the wall clock is reached). When set to 'adaptive' (the default), the distributed engine will try to adaptively scale the number of Workers based on tasks in the queue. This is str instead of bool type variable in case more complex adaptivity options are added in the future.

5.11.5 dask

class qcfractal.cli.qcfractal_manager.**DaskQueueSettings**(*, *interface: str = None*, *extra: List[str] = None*, *lsf_units: str = None*, ***kwargs*)

Settings for the Dask Cluster class. Values set here are passed directly into the Cluster objects based on the *cluster.scheduler* settings. Although many values are set automatically from other settings, there are some additional values such as *interface* and *extra* which are passed through to the constructor.

Valid values for this field are functions of your cluster.scheduler and no linting is done ahead of trying to pass these to Dask.

NOTE: The parameters listed here are a special exception for additional features Fractal has engineered or options which should be considered for some of the edge cases we have discovered. If you try to set a value which is derived from other options in the YAML file, an error is raised and you are told exactly which one is forbidden.

Please see the docs for the provider for more information.

Parameters

- **interface** (*str, Optional*) – Name of the network adapter to use as communication between the head node and the compute node.There are oddities of this when the head node and compute node use different ethernet adapter names and we have not figured out exactly which combination is needed between this and the poorly documented *ip* keyword which appears to be for Workers, but not the Client.
- **extra** (*List[str], Optional*) – Additional flags which are fed into the Dask Worker CLI startup, can be used to overwrite pre-configured options. Do not use unless you know exactly which flags to use.

- **lsf_units** (*str, Optional*) – Unit system for an LSF cluster limits (e.g. MB, GB, TB). If not set, the units are are attempted to be set from the *lsf.conf* file in the default locations. This does nothing if the cluster is not LSF

5.11.6 parsl

class qcfractal.cli.qcfractal_manager.**ParslQueueSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>'*, *_env_file_encoding: Optional[str] = None*, *_secrets_dir: Optional[Union[pathlib.Path, str]] = None*, ***, *executor: qcfractal.cli.qcfractal_manager.ParslExecutorSettings = ParslExecutorSettings(address=None)*, *provider: qcfractal.cli.qcfractal_manager.ParslProviderSettings = ParslProviderSettings(partition=None, launcher=None)*, ***values: Any*)

The Parsl-specific configurations used with the *common.adapter = parsl* setting. The parsl config is broken up into a top level *Config* class, an *Executor* sub-class, and a *Provider* sub-class of the *Executor*. Config -> Executor -> Provider. Each of these have their own options, and extra values fed into the ParslQueueSettings are fed to the *Config* level.

It requires both *executor* and *provider* settings, but will default fill them in and often does not need any further configuration which is handled by other settings in the config file.

Parameters

- **executor** (*ParslExecutorSettings*, Optional)
- **provider** (*ParslProviderSettings*, Optional)

executor

class qcfractal.cli.qcfractal_manager.**ParslExecutorSettings**(***, *address: str = None*, ***kwargs*)

Settings for the Parsl Executor class. This serves as the primary mechanism for distributing Workers to jobs. In most cases, you will not need to set any of these options, as several options are automatically inferred from other settings. Any option set here is passed through to the HighThroughputExecutor class of Parsl.

https://parsl.readthedocs.io/en/latest/stubs/parsl.executors.HighThroughputExecutor.html

NOTE: The parameters listed here are a special exception for additional features Fractal has engineered or options which should be considered for some of the edge cases we have discovered. If you try to set a value which is derived from other options in the YAML file, an error is raised and you are told exactly which one is forbidden.

Parameters **address** (*str, Optional*) – This only needs to be set in conditional cases when the head node and compute nodes use a differently named ethernet adapter.

An address to connect to the main Parsl process which is reachable from the network in which Workers will be running. This can be either a hostname as returned by hostname or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to identify what addresses are reachable from compute nodes.

provider

class qcfractal.cli.qcfractal_manager.**ParslProviderSettings**(*, *partition: str = None*, *launcher: qcfractal.cli.qcfractal_manager.ParslLauncherSetti = None*, ***kwargs*)

Settings for the Parsl Provider class. Valid values for this field depend on your choice of cluster.scheduler and are defined in the Parsl docs for the providers with some minor exceptions. The initializer function for the Parsl settings will indicate which

NOTE: The parameters listed here are a special exception for additional features Fractal has engineered or options which should be considered for some of the edge cases we have discovered. If you try to set a value which is derived from other options in the YAML file, an error is raised and you are told exactly which one is forbidden.

SLURM: https://parsl.readthedocs.io/en/latest/stubs/parsl.providers.SlurmProvider.html PBS/Torque/Moab: https://parsl.readthedocs.io/en/latest/stubs/parsl.providers.TorqueProvider.html SGE (Sun GridEngine): https://parsl.readthedocs.io/en/latest/stubs/parsl.providers.GridEngineProvider.html

Parameters

- **partition** (*str, Optional*) – The name of the cluster.scheduler partition being submitted to. Behavior, valid values, and evenits validity as a set variable are a function of what type of queue scheduler your specific cluster has (e.g. this variable should NOT be present for PBS clusters). Check with your Sys. Admins and/or your cluster documentation.
- **launcher** (`ParslLauncherSettings`, Optional) – The Parsl Launcher to use with your Provider. If left to `None`, defaults are assumed (check the Provider's defaults), otherwise this should be a dictionary requiring the option `launcher_class` as a str to specify which Launcher class to load, and the remaining settings will be passed on to the Launcher's constructor.

5.12 Configuration for High-Performance Computing

High-performance computing (HPC) clusters are designed to complete highly-parallel tasks in a short time. Properly leveraging such clusters requires utilizing large numbers of compute nodes at the same time, which requires special configurations for the QCFractal manager. This part of the guide details several routes for configuring HPC clusters to use either large numbers tasks that each use only a single node, or deploying a smaller number of tasks that use multiple nodes.

Note: This guide is currently limited to using the Parsl adapter and contains some configuration options which do not work with other adapters.

5.12.1 Many Nodes per Job, Single Node per Application

The recommended configuration for a QCFractal manager to use multi-node Jobs with tasks limited to a single node is launch many workers for a single Job.

The Parsl adapter deploys a single ‘’manager” per Job and uses the HPC system's MPI task launcher to deploy the Parsl executors on to the compute nodes. Each “executor” will run a single Python process per QCEngine worker and can run more than one worker per node. The `manager` will run on the login or batch node (depending on the cluster's configuration) once the Job is started and will communicate to the workers using Parsl's ØMQ messaging protocol. The QCFractal QueueManager will connect to the Parsl manager for each Job.

See the example page for details on how to configure Parsl for your system. The configuration setting `common.nodes_per_job` defines the ability to make multi-node allocation requests to a scheduler via an Adapter.

5.12.2 Many Nodes per Job, More than One Node per Application

The recommended configuration for using node-parallel tasks is to have a single QCFractal worker running on the batch node, and using that worker to launch MPI tasks on the compute nodes. The differentiating aspect of deploying multi-node tasks is that the QCFractal Worker and QCEngine Python process will run on different nodes than the quantum chemistry code.

The Parsl implementation for multi-node jobs will place a Parsl single executor and interchange on the login/batch node. The Parsl executor will launch a number of workers (as separate Python processes) equal to the number of nodes per Job divided by the number of nodes per Task. The worker will call the MPI launch system to place quantum-chemistry calculations on the compute nodes of the clusters.

See the example page for details on how to configure Parsl for your system.

5.13 Queue Manager Example YAML Files

The primary way to set up a *Manager* is to setup a YAML config file. This page provides helpful config files which mostly can be just copied and used in place (filling in things like ***username** and **password*** as needed.)

The full documentation of every option and how it can be used can be found in *the Queue Manager's API*.

For these examples, the `username` will always be “Foo” and the `password` will always be “b4R” (which are just placeholders and not valid). The `manager_name` variable can be any string and these examples provide some descriptive samples. The more distinct the name, the better it is to see its status on the *Server*.

5.13.1 SLURM Cluster, Dask Adapter with additional options

This example is similar to the *example on the start page for Managers*, but with some additional options such as connecting back to a central Fractal instance and setting more cluster-specific options. Again, this starts a manager with a dask *Adapter*, on a SLURM cluster, consuming 1 CPU and 8 GB of ram, targeting a Fractal Server running on that cluster, and using the SLURM partition `default`, save the following YAML config file:

```
common:
 adapter: dask
 tasks_per_worker: 1
 cores_per_worker: 1
 memory_per_worker: 8

server:
```

(continues on next page)

(continued from previous page)

```
 fractal_uri: "localhost:7777"
 username: Foo
 password: b4R

manager:
 manager_name: "SlurmCluster_OneDaskTask"

cluster:
 scheduler: slurm
 walltime: "72:00:00"

dask:
 queue: default
```

5.13.2 Multiple Tasks, 1 Cluster Job

This example starts a max of 1 cluster *Job*, but multiple *tasks*. The hardware will be consumed uniformly by the *Worker*. With 8 cores, 20 GB of memory, and 4 tasks; the *Worker* will provide 2 cores and 5 GB of memory to compute each *Task*. We set `common.max_workers` to 1 to limit the number of *Workers* and *Jobs* which can be started. Since this is SLURM, the `squeue` information will show this user has run 1 `sbatch` jobs which requested 4 cores and 20 GB of memory.

```
common:
 adapter: dask
 tasks_per_worker: 4
 cores_per_worker: 8
 memory_per_worker: 20
 max_workers: 1

server:
 fractal_uri: "localhost:7777"
 username: Foo
 password: b4R

manager:
 manager_name: "SlurmCluster_MultiDask"

cluster:
 scheduler: slurm
 walltime: "72:00:00"

dask:
 queue: default
```

5.13.3 Testing the Manager Setup

This will test the *Manager* to make sure it's setup correctly, and does not need to connect to the *Server*, and therefore does not need a `server` block. It will still however submit *jobs*.

```
common:
 adapter: dask
 tasks_per_worker: 2
 cores_per_worker: 4
 memory_per_worker: 10

manager:
 manager_name: "TestBox_NeverSeen_OnServer"
 test: True
 ntests: 5

cluster:
 scheduler: slurm
 walltime: "01:00:00"

dask:
 queue: default
```

5.13.4 Running commands before work

Suppose there are some commands you want to run *before* starting the *Worker*, such as starting a Conda environment, or setting some environment variables. This lets you specify that. For this, we will run on a Sun Grid Engine (SGE) cluster, start a conda environment, and load a module.

An important note about this one, we have now set `max_workers` to something larger than 1. Each *Job* will still request 16 cores and 256 GB of memory to be evenly distributed between the 4 *tasks*, however, the *Adapter* will **attempt to start 5 independent** *jobs*, for a total of 80 cores, 1.280 TB of memory, distributed over 5 *Workers* collectively running 20 concurrent *tasks*. If the *Scheduler* does not allow all of those *jobs* to start, whether due to lack of resources or user limits, the *Adapter* can still start fewer *jobs*, each with 16 cores and 256 GB of memory, but *Task* concurrency will change by blocks of 4 since the *Worker* in each *Job* is configured to handle 4 *tasks* each.

```
common:
 adapter: dask
 tasks_per_worker: 4
 cores_per_worker: 16
 memory_per_worker: 256
 max_workers: 5

server:
 fractal_uri: localhost:7777
 username: Foo
 password: b4R

manager:
 manager_name: "GridEngine_OpenMPI_DaskWorker"
 test: False

cluster:
 scheduler: sge
 task_startup_commands:
     - module load mpi/gcc/openmpi-1.6.4
```

(continues on next page)

(continued from previous page)

```
    - conda activate qcfmanager
 walltime: "71:00:00"

dask:
 queue: free64
```

5.13.5 Additional Scheduler Flags

A *Scheduler* may ask you to set additional flags (or you might want to) when submitting a *Job*. Maybe it's a Sys. Admin enforced rule, maybe you want to pull from a specific account, or set something not interpreted for you in the *Manager* or *Adapter* (do tell us though if this is the case). This example sets additional flags on a PBS cluster such that the final *Job* launch file will have `#PBS {my headers}`.

This example also uses Parsl and sets a scratch directory.

```
common:
 adapter: parsl
 tasks_per_worker: 1
 cores_per_worker: 6
 memory_per_worker: 64
 max_workers: 5
 scratch_directory: "$TMPDIR"

server:
 fractal_uri: localhost:7777
 username: Foo
 password: b4R
 verify: False

manager:
 manager_name: "PBS_Parsl_MyPIGroupAccount_Manger"

cluster:
 node_exclusivity: True
 scheduler: pbs
 scheduler_options:
    - "-A MyPIsGroupAccount"
 task_startup_commands:
    - conda activate qca
    - cd $WORK
 walltime: "06:00:00"

parsl:
 provider:
  partition: normal_q
  cmd_timeout: 30
```

5.13.6 Single Job with Multiple Nodes and Single-Node Tasks with Parsl Adapter

Leadership platforms prefer or require more than one node per Job request. The following configuration will request a Job with 256 nodes and place one Worker on each node.

```
common:
    adapter: parsl
    tasks_per_worker: 1
    cores_per_worker: 64  # Number of cores per compute node
    max_workers: 256  # Maximum number of workers deployed to compute nodes
    nodes_per_job: 256

cluster:
    node_exclusivity: true
    task_startup_commands:
        - module load miniconda-3/latest  # You will need to load the Python␣
↪environment on startup
        - source activate qcfractal
        - export KMP_AFFINITY=disable  # KNL-related issue. Needed for multithreaded␣
↪apps
        - export PATH=~/software/psi4/bin:$PATH  # Points to psi4 compiled for␣
↪compute nodes
    scheduler: cobalt  # Varies depending on supercomputing center

parsl:
    provider:
        queue: default
        launcher:  # Defines the MPI launching function
            launcher_class: AprunLauncher
            overrides: -d 64  # Option for XC40 machines, allows workers to access 64␣
↪threads
        init_blocks: 0
        min_blocks: 0
        account: CSC249ADCD08
        cmd_timeout: 60
        walltime: "3:00:00"
```

Consult the Parsl configuration docs for information on how to configure the Launcher and Provider classes for your cluster.

5.13.7 Single Job with Multiple, Node-Parallel Tasks with Parsl Adapter

Running MPI-parallel tasks requires a similar configuration to the multiple nodes per job for the manager and also some extra work in defining the qcengine environment. The key difference that sets apart managers for node-parallel applications is that that `nodes_per_job` is set to more than one and Parsl uses `SimpleLauncher` to deploy a Parsl executor onto the batch/login node once a job is allocated.

```
common:
    adapter: parsl
    tasks_per_worker: 1
    cores_per_worker: 16  # Number of cores used on each compute node
    max_workers: 128
    memory_per_worker: 180  # Summary for the amount per compute node
    nodes_per_job: 128
    nodes_per_task: 2  # Number of nodes to use for each task
    cores_per_rank: 1  # Number of cores to each of each MPI rank
```

(continues on next page)

(continued from previous page)

```
cluster:
    node_exclusivity: true
    task_startup_commands:
        - module load miniconda-3/latest
        - source activate qcfractal
        - export PATH="/soft/applications/nwchem/6.8/bin/:$PATH"
        - which nwchem
    scheduler: cobalt

parsl:
    provider:
        queue: default
        launcher:
            launcher_class: SimpleLauncher
        init_blocks: 0
        min_blocks: 0
        account: CSC249ADCD08
        cmd_timeout: 60
        walltime: "0:30:00"
```

The configuration that describes how to launch the tasks must be written at a `qcengine.yaml` file. See QCEngine docs for possible locations to place the `qcengine.yaml` file and full descriptions of the configuration option. One key option for the `qcengine.yaml` file is the description of how to launch MPI tasks, `mpiexec_command`. For example, many systems use `mpirun` (e.g., OpenMPI). An example configuration a Cray supercomputer is:

```
all:
  hostname_pattern: "*"
  scratch_directory: ./scratch  # Must be on the global filesystem
  is_batch_node: True  # Indicates that `aprun` must be used for all QC code␣
↪invocations
  mpiexec_command: "aprun -n {total_ranks} -N {ranks_per_node} -C -cc depth --env␣
↪CRAY_OMP_CHECK_AFFINITY=TRUE --env OMP_NUM_THREADS={cores_per_rank} --env MKL_NUM_
↪THREADS={cores_per_rank}
  -d {cores_per_rank} -j 1"
  jobs_per_node: 1
  ncores: 64
```

Note that there are several variables in the `mpiexec_command` that describe how to insert parallel configurations into the command: `total_ranks`, `ranks_per_node`, and `cores_per_rank`. Each of these values are computed based on the number of cores per node, the number of nodes per application and the number of cores per MPI rank, which are all defined in the Manager settings file.

5.14 Queue Manager Frequent Questions and Issues

This page documents some of the frequent questions and issues we see with the Queue Managers. If this page and none of the other documentation pages have answered your question, please ask on GitHub or join our Slack group to get assistance.

5.14.1 Common Questions

How do I get more information from the Manger?

Turn on `verbose` mode, either add the `-v` flag to the CLI, or set the `common.verbose` to `True` in the YAML file. Setting this flag will produce much more detailed information. This sets the loggers to `DEBUG` level.

In the future, we may allow for different levels of increased verbosity, but for now there is only the one level.

Can I start more than one Manager at a time?

Yes. This is often done if you would like to create multiple *task tags* that have different resource requirements or spin up managers that can access different resources. Check with your cluster administrators though to find out their policy on multiple processes running on the clusters head node.

You can reuse the same config file, just invoke the CLI again.

Can I connect to a Fractal Server besides MolSSI's?

Yes! Just change the `server.fractal_uri` argument.

Can I connect to more than one Fractal Server

Yes and No. Each *Manager* can only connect to a single *Fractal Server*, but you can start multiple managers with different config files pointing to different *Fractal Servers*.

How do I help contribute compute time to the MolSSI database?

Join our Slack group! We would love to talk to you and help get you contributing as well!

I have this issue, here is my config file...

Happy to look at it! We only ask that you please **remove the password from the config file before posting it.** If we see a password, we'll do our best to delete it, but that does not ensure someone did not see it.

5.14.2 Common Issues

This documents some of the common issues we see.

Jobs are quickly started and die without error

We see this problem with Dask often and the most common case is the head node (landing node, login node, etc.) has an ethernet adapter with a different name than the compute nodes. You can check this by running the command `ip addr` on both the head node and a compute node (either through an interactive job or a job which writes the output of that command to a file).

You will see many lines of output, but there should be a block that looks like the following:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
3: eno49.4010@eno49: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state␣
↪UP qlen 1000
    inet 10.XX.Y.Z/16 brd 10.XX.255.255 scope global eno49.4010
4: eno49.4049@eno49: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state␣
↪UP qlen 1000
    inet 198.XX.YYY.ZZZ/24 brd 198.XX.252.255 scope global eno49.4049
```

the `XX`, `YYY` and `ZZZ` will have values corresponding to your cluster's configuration. There are a few critical items:

- The headers (`lo`, `eno.49...`, yours will be different) and the addresses where he `XX` placeholders are.
- Ignore the `lo` adapter, every machine should have one.
- The head node should have a `inet` that looks like a normal IP address, and another one which looks like it has a `10.something` IP address.
- The compute node will likely have an adapter which is only the `10.something`.
- These `10.something` IP addresses are often intranet communication only, meaning the compute nodes cannot reach the broader internet

The name of the ethernet adapter housing the `10.something` will be *different* on the head node and the compute node.

In this case, in your YAML file, add a line in `dask` called `interface` and set it to the name of the adapter which is shared. So for this it would be:

```
dask:
  interface: "eno49.4049"
```

plus all the rest of your YAML file. You can safely ignore the bit after the `@` sign.

If there isn't a shared adapter name, try this instead:

```
dask:
  ip: "10.XX.Y.Z"
```

Replace the `.XX.Y.Z` with the code which has the intranet IP of the *head node*. This option acts as a pass through to the Dask *Worker* call and tells the worker to try and connect to the head node at that IP address.

If that still doesn't work, contact us. We're working to make this less manual and difficult in the future.

Other variants:

- "My jobs start and stop instantly"

- "My jobs restart forever"

My Conda Environments are not Activating

You likely have to `source` the Conda `profile.d` again first. See also https://github.com/conda/conda/issues/8072

This can also happen during testing where you will see command-line based binaries (like Psi4) pass, but Python-based codes (like RDKit) fail saying complaining about an import error. On cluster compute nodes, this often manifests as the `$PATH` variable being passed from your head node correctly to the compute node, but then the Python imports cannot be found because the Conda environment is not set up correctly.

This problem is obfuscated by the fact that *workers* such as Dask Workers can still start initially despite being a Python code themselves. Many *adapters* will start their programs using the absolute Python binary path which gets around the incomplete Conda configuration. **We strongly recommend you do not try setting the absolute Python path** in your scripts to get around this, and instead try to `source` the Conda `profile.d` first. For example, you might need to add something like this to your YAML file (change paths/environment names as needed):

```
cluster:
    task_startup_commands:
        - source ~/miniconda3/etc/profile.d/conda.sh
        - conda activate qcfractal
```

Other variants:

- "Tests from one program pass, but others don't"
- "I get errors about unable to find program, but its installed"
- "I get path and/or import errors when testing"

My jobs appear to be running, but only one (or few) workers are starting

If the jobs appear to be running (and the Manager is reporting they return successfully), a few things may be happening.

- If jobs are completing very fast, the *Adapter* may not feel like it needs to start more *workers*, which is fine.
- (Not recommended, use for debug only) Check your `manger.max_queued_tasks` arg to pull more *tasks* from the *Server* to fill the jobs you have started. This option is usually automatically calculated based on your `common.tasks_per_worker` and `common.max_workers` to keep all *workers* busy and still have a buffer.

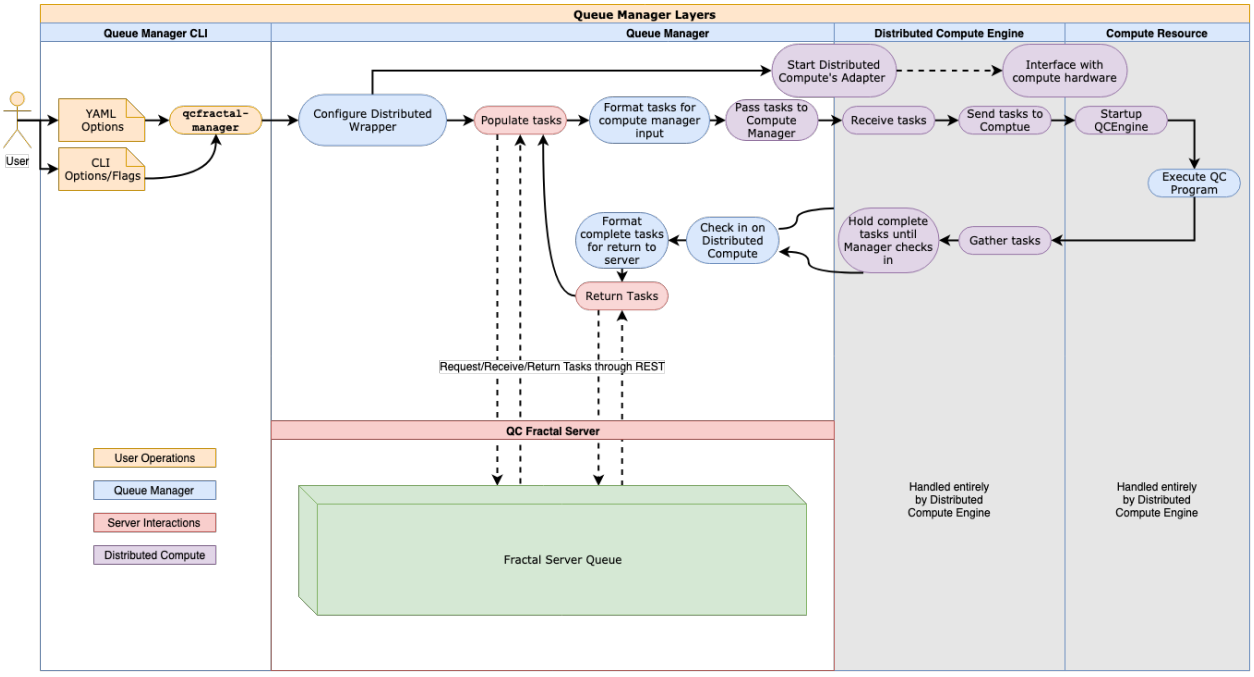
5.15 Detailed Manager Information

This page documents all the internals of the *Managers* in depth and is **not** intended for the general user nor should be required for setting up and running them. This page is for those who are interested in the inner workings and detailed flow of the how the *Manager* interacts with the *Server*, the *Adapter*, the *Scheduler*, and what it is abstracting away from the user.

Since this is intended to be a deep dive into mechanics, please let us know if something is missing or unclear and we can expand this page as needed.

5.15.1 Manager Flowchart

The Queue Manager's interactions with the Fractal Server, the Distributed Compute Engine, the physical Compute Hardware, and the user are shown in the following diagram.



Server Documentation

Configuring and running the Server from the CLI and Config Files

- *Fractal Server Init*
- *Fractal Server Config*
- *Fractal Server Start*
- *Fractal Server User*
- *Fractal Server Upgrade*

5.16 Fractal Server Init

The sub-command for the `qcfractal-server` CLI which initializes a new server instance, including configuring the PostgreSQL database if it is not setup yet.

5.16.1 Command Invocation

```
qcfractal-server init [<options>]
```

5.16.2 Command Description

This command will attempt to do the following actions for the user in default mode (no args):

- Create the *QCFractal Config directory*
- Create a blank *Fractal Config file* (assumes defaults)
- Create the folders for housing the PostgreSQL database file, which will be the home of Fractal's data.
- Initialize PostgreSQL's service at the database location from above
- Start the PostgreSQL server
- Populate the database tables and finalize everything for Fractal's operation

In most cases, the user should not have to change any configurations if they are the system owners or admins. However, if users want to do something different, they can write their own *Config File* and change the settings though the CLI to start the server.

5.16.3 Options

This is a set of GLOBAL level options which impact where the ``init`` command looks, and how it interacts with the config file

`--overwrite` Control whether the rest of the settings overwrite an existing config file in the *QCFractal Config directory*

`--base-folder [<folder>]` The QCFractal base directory to attach to. Default: `~/.qca/qcfractal`

This set of options pertain to the PostgreSQL database itself and translate to the `database` header in the *Fractal Server Config*.

`--db-port [<port>]` The PostgreSQL default port, Default `5432`

`--db-host [<host>]` Default location for the Postgres server. If not `localhost`, Fractal command lines cannot manage the instance. and will have to be configured in the *Config File*. Default: `localhost`

`--db-username [<user>]` The postgres username to default to. **Planned Feature - Currently inactive**.

`--db-password [<password>]` The postgres password for the give user. **Planned Feature - Currently inactive**.

`--db-directory [<dir_path>]` "The physical location of the QCFractal instance data, defaults to the root *Config directory*.

`--db-default-database [<db_name>]` The default database to connect to. Typically used if you already have a Fractal Database set up or you want to use a different name for the database besides the default. Default `qcfractal_default`.

--db-logfile [<logfile>] The logfile to write postgres logs. Default `qcfractal_postgres.log`.

--db-own (True|False) If own is True, Fractal will control the database instance. If False Postgres will expect a booted server at the database specification. Default `True`

The settings below here pertain to the Fractal Server and translate to the `fractal` header in the *Fractal Server Config*.

--name [<name>] The Fractal server default name. Controls how the server presents itself to connected clients. Default `QCFractal Server`

--port [<port>] The Fractal default port. This is the port which Fractal listens to for client connections (and for the URI). This is *separate* from the `--db-port` which is the port that PostgreSQL database is listening for. In general, these should be separate. Default `7777`.

--compres-response (True|False) Compress REST responses or not, should be True unless behind a proxy. Default `True`.

--allow-read (True|False) Always allows read access to record tables. Default `True`

--security [<security_string>] Optional security features. Not set by default.

--query-limit [<int_limit>] The maximum number of records to return per query. Default `1000`

--logfile [<log>] The logfile the Fractal Server writes to. Default `qcfractal_server.log`

--service-frequency [<frequency>] The frequency to update the Fractal services. Default `60`

--max-active-services [<max-services>] The maximum number of concurrent active services. Default `20`

--heartbeat-frequency [<heartbeat>] The frequency (in seconds) to check the heartbeat of *Managers*. Default `1800`

5.17 Fractal Server Config

This page documents the valid options for the YAML file inputs to the *Config File*. This first section outlines each of the headers (top level objects) and a description for each one. The final file will look like the following:

```yaml
common:
    option_1: value_for1
    another_opt: 42
server:
    option_for_server: "some string"
```

5.17.1 Command Invocation

```
qcfractal-server config [<options>]
```

5.17.2 Command Description

Show the current config file at an optional location.

Looks in the default location if no arg is provided

5.17.3 Options

--base-folder [<folder>] The QCFractal base directory to attach to. Default: `~/.qca/qcfractal`

5.17.4 Config File Complete Options

The valid top-level YAML headers are the parameters of the `FractalConfig` class.

class qcfractal.config.**FractalConfig**(*, *base_folder: str = '/home/docs/.qca/qcfractal'*, *database: qcfractal.config.DatabaseSettings = DatabaseSettings(port=5432, host='localhost', username=None, password=None, directory=None, database_name='qcfractal_default', logfile='qcfractal_postgres.log', own=True)*, *view: qcfractal.config.ViewSettings = ViewSettings(enable=True, directory=None)*, *fractal: qcfractal.config.FractalServerSettings = FractalServerSettings(name='QCFractal Server', port=7777, compress_response=True, allow_read=True, security=None, query_limit=1000, logfile='qcfractal_server.log', loglevel='info', cprofile=None, service_frequency=60, max_active_services=20, heartbeat_frequency=1800, log_apis=False, geo_file_path=None)*)

Top level configuration headers and options for a QCFractal Configuration File

Parameters

- **base_folder** (*str, Default: /home/docs/.qca/qcfractal*) – The QCFractal base instance to attach to. Default will be your home directory
- **database** (*DatabaseSettings*, Optional)
- **view** (*ViewSettings*, Optional)
- **fractal** (*FractalServerSettings*, Optional)

database

class qcfractal.config.**DatabaseSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>'*, *_env_file_encoding: Optional[str] = None*, *_secrets_dir: Optional[Union[pathlib.Path, str]] = None*, ***, *port: int = 5432*, *host: str = 'localhost'*, *username: str = None*, *password: str = None*, *directory: str = None*, *database_name: str = 'qcfractal_default'*, *logfile: str = 'qcfractal_postgres.log'*, *own: bool = True*)

Postgres Database settings

Parameters

- **port** (*int, Default: 5432*) – The postgresql default port
- **host** (*str, Default: localhost*) – Default location for the postgres server. If not localhost, qcfractal command lines cannot manage the instance.
- **username** (*str, Optional*) – The postgres username to default to.
- **password** (*str, Optional*) – The postgres password for the give user.
- **directory** (*str, Optional*) – The physical location of the QCFractal instance data, defaults to the root folder.
- **database_name** (*str, Default: qcfractal_default*) – The database name to connect to.
- **logfile** (*str, Default: qcfractal_postgres.log*) – The logfile to write postgres logs.
- **own** (*bool, Default: True*) – If own is True, QCFractal will control the database instance. If False Postgres will expect a booted server at the database specification.

fractal

class qcfractal.config.**FractalServerSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>'*, *_env_file_encoding: Optional[str] = None*, *_secrets_dir: Optional[Union[pathlib.Path, str]] = None*, ***, *name: str = 'QCFractal Server'*, *port: int = 7777*, *compress_response: bool = True*, *allow_read: bool = True*, *security: str = None*, *query_limit: int = 1000*, *logfile: str = 'qcfractal_server.log'*, *loglevel: str = 'info'*, *cprofile: str = None*, *service_frequency: int = 60*, *max_active_services: int = 20*, *heartbeat_frequency: int = 1800*, *log_apis: bool = False*, *geo_file_path: str = None*)

Fractal Server settings

Parameters

- **name** (*str, Default: QCFractal Server*) – The QCFractal server default name.
- **port** (*int, Default: 7777*) – The QCFractal default port.
- **compress_response** (*bool, Default: True*) – Compress REST responses or not, should be True unless behind a proxy.
- **allow_read** (*bool, Default: True*) – Always allows read access to record tables.
- **security** (*str, Optional*) – Optional user authentication. Specify 'local' to enable authentication through locally stored usernames. User permissions may be manipulated through the `qcfractal-server user` CLI.
- **query_limit** (*int, Default: 1000*) – The maximum number of records to return per query.
- **logfile** (*str, Default: qcfractal_server.log*) – The logfile to write server logs.
- **loglevel** (*str, Default: info*) – Level of logging to enable (debug, info, warning, error, critical)
- **cprofile** (*str, Optional*) – Enable profiling via cProfile, and output cprofile data to this path
- **service_frequency** (*int, Default: 60*) – The frequency to update the QCFractal services.

- **max_active_services** (*int, Default: 20*) – The maximum number of concurrent active services.
- **heartbeat_frequency** (*int, Default: 1800*) – The frequency (in seconds) to check the heartbeat of workers.
- **log_apis** (*bool, Default: False*) – True or False. Store API access in the Database. This is an advanced option for servers accessed by external users through QCPortal.
- **geo_file_path** (*str, Optional*) – Geoip2 cites file path (.mmdb) for resolving IP addresses. Defaults to [base_folder]/GeoLite2-City.mmdb

view

class qcfractal.config.**ViewSettings**(*_env_file: Optional[Union[pathlib.Path, str]] = '<object object>'*, *_env_file_encoding: Optional[str] = None*, *_secrets_dir: Optional[Union[pathlib.Path, str]] = None*, ***, *enable: bool = True*, *directory: str = None*)

HDF5 view settings

Parameters

- **enable** (*bool, Default: True*) – Enable frozen-views.
- **directory** (*str, Optional*) – Location of frozen-view data. If None, defaults to base_folder/views.

5.18 Fractal Server Start

The sub-command for the `qcfractal-server` CLI which starts the Fractal server instance

5.18.1 Command Invocation

```
qcfractal-server start [<options>]
```

5.18.2 Command Description

This command will attempt to do the following actions for the user in default mode (no args):

- Read the *QCFractal Config directory*
- Read the config file in that directory
- Connect to the previously created Fractal database created in the PostgreSQL service (see *Fractal Server Init*).
- Start Fractal's periodic services.
- Create and provide SSL certificates.

The options for the database and starting local compute on the same resources as the server can be controlled through the flags below. Also see all the config file options in *Config File*.

5.18.3 Options

--base-folder [<folder>] The QCFractal base directory to attach to. Default: `~/.qca/qcfractal`

--port [<port>] The Fractal default port. This is the port which Fractal listens to for client connections (and for the URI). This is *separate* from the *the port that PostgreSQL database is listening for. In general, these should be separate. Default ``7777`*.

--logfile [<log>] The logfile the Fractal Server writes to. Default `qcfractal_server.log`

--database-name [<db_name>] The database to connect to, defaults to the default database name. Default `qcfractal_default`

--server-name [<server_name>] The Fractal server default name. Controls how the server presents itself to connected clients. Default `QCFractal Server`

--start-periodics (True|False) Expert Level Flag Only Warning! Can disable periodic update (services, heartbeats) if False. Useful when running behind a proxy. Default `True`

--disable-ssl (False|True) Disables SSL if present, if `False` a SSL cert will be created for you. Default `False`

--tls-cert [<tls_cert_str>] Certificate file for TLS (in PEM format)

--tls-key [<tls_key_str>] Private key file for TLS (in PEM format)

--local-manager [<int>] Creates a local pool QueueManager attached to the server using the number of threads specified by the arg. If this flag is set and no number is provided, 1 (one) thread will be spun up and running locally. If you expect *Fractal Managers* to connect to this server, then it is unlikely you need this. Related, if no compute is expected to be done on this server, then it is unlikely this will be needed.

5.19 Fractal Server User

The sub-command for the `qcfractal-server` CLI which manages user permissions and passwords.

5.19.1 Command Invocation

```
qcfractal-server user [<options>]
```

5.19.2 Top-level Options

--base-folder [<folder>] The QCFractal base directory to attach to. Default: `~/.qca/qcfractal`.

5.19.3 Subcommand Summary

The `qcfractal-server user` CLI allows for manipulation of users through four subcommands:

- `add`: Add a new user.
- `show`: Display a user's permissions.
- `modify`: Change a user's permissions or password.
- `remove`: Delete a user.

5.19.4 Add Subcommand

Command Invocation

```
qcfractal-server user add [<options>] <username>
```

Command Description

This command adds a new user, setting the user's password and permissions. The user must not already exist.

Arguments

<username> The username to add.

--password [<password>] The password for the user. If this option is not provided, a password will be generated and printed.

--permissions [<permissions>] Permissions for the user. Allowed values: `read`, `write`, `queue`, `compute`, `admin`. Multiple values are allowed. At least one value must be specified.

5.19.5 Show Subcommand

Command Invocation

```
qcfractal-server user show <username>
```

Command Description

This command prints the permissions for a given user.

Arguments

<username> The username for which to show permissions.

5.19.6 Modify Subcommand

Command Invocation

```
qcfractal-server user modify [<options>] <username>
```

Command Description

This command modifys a user's permissions or password.

Arguments

<username> The username to modfiy.

--password [<password>] Change the user's password to a given string. This options excludes `--reset-password`.

--reset-password Change the user's password to an auto-generated value. The new password will be printed. This option excludes `--password`.

--permissions [<permissions>] Change the user's permissions to the given set. Allowed values: `read`, `write`, `queue`, `compute`, `admin`. Multiple values are allowed. See *User Permissions* for more information.

5.19.7 Remove Subcommand

Command Invocation

```
qcfractal-server user remove <username>
```

Command Description

This command removes a user.

Arguments

<username> The username to remove.

5.19.8 User Permissions

Five permission types are available:

- `read` allows read access to existing records.
- `write` allows write access to existing records and the ability to add new records.
- `compute` allows enqueuing new *Tasks*.
- `queue` allows for consumption of compute *Tasks*. This permission is intended for use by a *Manager*.
- `admin` allows all permissions.

5.20 Fractal Server Upgrade

The sub-command for the `qcfractal-server` CLI which allows in-place upgrade of Fractal Databases to newer versions through SQLAlchemy Alembic.

5.20.1 Command Invocation

```
qcfractal-server upgrade [<options>]
```

5.20.2 Command Description

This command will attempt to upgrade an existing Fractal Database (stored in PostgreSQL) to a new version based on the currently installed Fractal software version. Not every version of Fractal updates the database, so this command will only need to be run when you know the database has changed (or attempting to start it tells you to).

This command will attempt to do the following actions for the user in default mode (no args):

- Read the database location from your *Config File* in the default location (can be controlled)
- Determine the upgrade paths from your existing version to the version known by Alembic (update information is shipped with the Fractal software)
- Stage update
- Commit update if no errors found

You will then need to start the server again through *Fractal Server Start* to bring the server back online.

Caveat: This command will **not** initialize the Fractal Database for you from nothing. The database must exist for this command to run.

5.20.3 Options

`--base-folder [<folder>]` The QCFractal base directory to attach to. Default: `~/.qca/qcfractal`

5.21 Server-side Dataset Views

Note: This is an experimental feature.

HDF5 views of Datasets may be stored on the server to improve query performance. To use views, first specify a path to store views in the *`qcfractal.config.ViewSettings`*.

Next, generate a view for the collection(s) of interest:

```
import qcfractal.interface as ptl
ds = ptl.get_collection("ReactionDataset", "S22")

# Note the server will look for views in the directory specified above,
# named {collection_id}.hdf5
view = ptl.collections.HDF5View(viewpath / f"{ds.data.id}.hdf5")
view.write(ds)
```

Finally, mark the collection as supporting views:

```
# Update the dataset to indicate a view is available
ds.__dict__["view_available"] = True
ds.save()

# Optionally, you may add a download URL for the view
ds.__dict__["view_url"] = "https://someserver.com/view.hdf5"
ds.save()
```

Developer Documentation

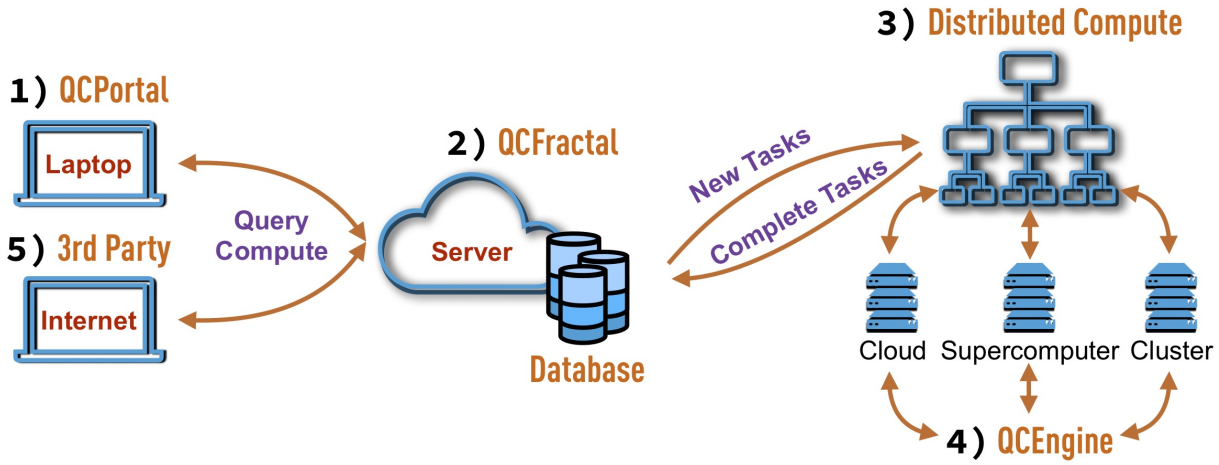
Contains in-depth developer documentation.

- *QCArchive Design*
- *Glossary*
- *Changelog*
- *Glossary*

5.22 QCArchive Design

The QCArchive software ecosystem consists of a series of Python modules that can either be used together or are useful standalone pieces to the computational molecular sciences community. This ecosystem is constructed to be used at single-user, small group, and multi-PI levels while retaining the ability to scale up to the needs of an entire community of scientist.

In each case, it is expected only a small number of users are required to understand the entire software stack and the primary interaction with the QCArchive ecosystem will be through the user front-end (QCPortal). After the persistence server instance (QCFractal) is instantiated with a distributed workflow system and compute the server should be able to maintain itself without user intervention. A diagram of how the ecosystem works in concert can be seen below:



5.22.1 1) QCPortal

- Hardware: Laptop
- Actor: User
- Primary Developer: MolSSI

QCPortal provides a Python-based user front-end experience for users who are interested in exploring data and executing new tasks. Exploration of computed data is augmented by the ability to generate graphs quickly and other representations to view the data in Jupyter notebooks and high-level abstractions are used to view and manipulate many individual tasks simultaneously. Querying of data and submission of new tasks occurs over the QCFractal REST API.

5.22.2 2) QCFractal

- Hardware: Persistent Server
- Actor: Power User
- Primary Developer: MolSSI

QCFractal is the primary persistent server that QCPortal communicates with and has several main duties:

- Maintain a database of all completed quantum chemistry results along with metadata that forms higher-level collections of results.
- Maintain a compute queue of all requested and completed tasks. Where each task is a single quantum chemistry result.
- Submit new tasks to distributed workflow engines and insert complete results into the database.
- Maintain high level compute workflows via the "Services" feature.

5.22.3 3) Distributed Compute

- Hardware: Persistent Server/Supercomputer
- Actor: Power User (can be separate from the Fractal Power Users)
- Primary Developer: Scientific and HPC Communities

The QCArchive project relies on a number of distributed compute workflow engines to enable a large variety of compute workloads. QCFractal will interact with each of these projects by submitting a series of tasks that do not have data or execution order dependence. The communication interfaces vary from Python-based API calls to REST API interfaces depending on the implementation details of the individual tools.

Current distributed compute backends are:

- Dask Distributed - Multi-node task graph scheduler built in Python.
- Parsl - High-performance workflows.
- Fireworks - Multi-site task scheduler built in Python with a central MongoDB server.

Pending backend implementations include:

- RADICAL Cybertools - Distributed task scheduler built for DOE and NSF compute resources.

- BOINC - High throughput volunteer computing task manager.
- Balsam - Task manager for a single compute resource (supercomputer) with tasks pulled from a central server.

The compute workers of each of these tools is executed in different ways. However, in each case the compute workers will distribute QCSchema inputs, call QCEngine, and receive a QCSchema output.

5.22.4 4) QCEngine

- Hardware: Local Cluster, Supercomputer, or Cloud Compute
- Actor: Power User

QCEngine is a stateless, lightweight wrapper around Quantum Chemistry programs so that these programs consistently accept and emit QCSchema. Depending on the underlying program QCEngine provides this uniform interface by either:

1) Calling the QCSchema IO functions that individual program have implemented.
2) Calling the Python-API of the program and modifying the input/output according to the QCSchema.
3) Writing a ASCII input file based on the input QCSchema, running the program, and parsing an ASCII output file into the QCSchema.

QCEngine also keeps track of the provenance of each task. This includes:

- A description of the hardware used (CPU, GPU, memory, etc).
- The total compute time and resources allocated to the run.
- The function and version of the program called.

5.22.5 5) 3rd Party Services

- Hardware: Laptop
- Actor: User/Power User
- Primary Developer: Computational Molecular Sciences Community

The QCFractal API is expected to have additional services attached by 3rd parties. These services can range from cross-reference data services to user website that visualize and interact with the data in a specific way,

5.23 QCFractal API

5.23.1 qcfractal Package

Main init function for qcfractal

Functions

storage_socket_factory(uri[, project_name, ...])	Factory for generating storage sockets.

storage_socket_factory

qcfractal.**storage_socket_factory**(*uri*, *project_name=''*, *logger=None*, *db_type=None*, ***kwargs*)

Factory for generating storage sockets. Spins up a given storage layer on request given common inputs.

Right now only supports MongoDB.

Parameters

- **uri** (*string*) – A URI to given database such as ("postgresql://localhost:5432",)
- **project_name** (*string*) – Name of the project
- **logger** (*logging.Logger, Optional, Default: None*) – Specific logger to report to
- **db_type** (*string, Optional, Default: 'sqlalchemy'*) – socket type, 'sqlalchemy'
- ****kwargs** – Additional keyword arguments to pass to the storage constructor

Classes

FractalServer(name, port, loop, ... [, ...])	
FractalSnowflake(max_workers, storage_uri, ...)	
FractalSnowflakeHandler(ncores)	
PostgresHarness(config, Any], ...)	
QueueManager(client, queue_client, logger, ...)	This object maintains a computational queue and watches for finished tasks for different queue backends.
TemporaryPostgres(database_name, tmpdir, ...)	

FractalServer

class qcfractal.**FractalServer**(*name: str = 'QCFractal Server', port: int = 7777, loop: IOLoop = None, compress_response: bool = True, security: Optional[str] = None, allow_read: bool = False, ssl_options: Union[bool, Dict[str, str]] = True, storage_uri: str = 'postgresql://localhost:5432', storage_project_name: str = 'qcfractal_default', query_limit: int = 1000, view_enabled: bool = False, view_path: Optional[str] = None, logfile_prefix: str = None, loglevel: str = 'info', log_apis: bool = False, geo_file_path: str = None, queue_socket: BaseAdapter = None, heartbeat_frequency: float = 1800, max_active_services: int = 20, service_frequency: float = 60, skip_storage_version_check=True*)

Bases: object

Methods Summary

add_exit_callback(callback, *args, **kwargs)	Adds additional callbacks to perform when closing down the server.
await_results()	A synchronous method for testing or small launches that awaits task completion before adding all queued results to the database and returning.
await_services([max_iter])	A synchronous method that awaits the completion of all services before returning.
check_manager_heartbeats()	Checks the heartbeats and kills off managers that have not been heard from.
client()	Builds a client from this server.
get_address([endpoint])	Obtains the full URI for a given function on the FractalServer.
list_current_tasks()	Provides a list of tasks currently in the queue along with the associated keys.
list_managers([status, name])	Provides a list of managers associated with the server both active and inactive.
start([start_loop, start_periodics])	Starts up the IOLoop and periodic calls.
stop([stop_loop])	Shuts down the IOLoop and periodic updates.
update_public_information()	Updates the public information data
update_server_log()	Updates the servers internal log
update_services()	Runs through all active services and examines their current status.
update_tasks()	Pulls tasks from the queue_adapter, inserts them into the database, and fills the queue_adapter with new tasks.

Methods Documentation

add_exit_callback(*callback*, **args*, ***kwargs*)
Adds additional callbacks to perform when closing down the server.

Parameters

- **callback** (*callable*) – The function to call at exit
- ***args** – Arguments to call with the function.
- ****kwargs** – Kwargs to call with the function.

await_results() → bool
A synchronous method for testing or small launches that awaits task completion before adding all queued results to the database and returning.

Returns Return True if the operation completed successfully

Return type bool

await_services(*max_iter: int = 10*) → bool
A synchronous method that awaits the completion of all services before returning.

Parameters **max_iter** (*int, optional*) – The maximum number of service iterations the server will run through. Will terminate early if all services have completed.

Returns Return True if the operation completed successfully

Return type bool

check_manager_heartbeats() → None
Checks the heartbeats and kills off managers that have not been heard from.

client()
Builds a client from this server.

get_address(*endpoint: Optional[str] = None*) → str
Obtains the full URI for a given function on the FractalServer.

Parameters **endpoint** (*Optional[str], optional*) – Specifies a endpoint to provide the URI for. If None returns the server address.

Returns The endpoint URI

Return type str

list_current_tasks() → List[Any]
Provides a list of tasks currently in the queue along with the associated keys.

Returns **ret** – All tasks currently still in the database

Return type list of tuples

list_managers(*status: Optional[str] = None*, *name: Optional[str] = None*) → List[Dict[str, Any]]
Provides a list of managers associated with the server both active and inactive.

Parameters

- **status** (*Optional[str], optional*) – Filters managers by status.
- **name** (*Optional[str], optional*) – Filters managers by name

Returns The requested Manager data.

Return type List[Dict[str, Any]]

start(*start_loop: bool = True*, *start_periodics: bool = True*) → None
Starts up the IOLoop and periodic calls.

Parameters

- **start_loop** (*bool, optional*) – If False, does not start the IOLoop
- **start_periodics** (*bool, optional*) – If False, does not start the server periodic updates such as Service iterations and Manager heartbeat checking.

stop(*stop_loop: bool = True*) → None
Shuts down the IOLoop and periodic updates.

Parameters **stop_loop** (*bool, optional*) – If False, does not shut down the IOLoop. Useful if the IOLoop is externally managed.

update_public_information() → None
Updates the public information data

update_server_log() → Dict[str, Any]
Updates the servers internal log

update_services() → int
Runs through all active services and examines their current status.

update_tasks() → bool
Pulls tasks from the queue_adapter, inserts them into the database, and fills the queue_adapter with new tasks.

Returns Return True if the operation completed successfully

Return type bool

FractalSnowflake

class qcfractal.**FractalSnowflake**(*max_workers: Optional[int] = 2*, *storage_uri: Optional[str] = None*, *storage_project_name: str = 'temporary_snowflake'*, *max_active_services: int = 20*, *logging: Union[bool, str] = False*, *start_server: bool = True*, *reset_database: bool = False*)
Bases: qcfractal.server.FractalServer

Methods Summary

client()	Builds a client from this server.
stop()	Shuts down the Snowflake instance.

Methods Documentation

client()
Builds a client from this server.

stop() → None
Shuts down the Snowflake instance. This instance is not recoverable after a stop call.

FractalSnowflakeHandler

class qcfractal.**FractalSnowflakeHandler**(*ncores: int = 2*)
Bases: object

Attributes Summary

logfilename	

Methods Summary

client()	Builds a client from this server.
get_address([endpoint])	Obtains the full URI for a given function on the FractalServer.
restart([timeout])	Restarts the current FractalSnowflake instances and destroys all data in the process.
show_log([nlines, clean, show])	Displays the FractalSnowflakes log data.
start([timeout])	Stop the current FractalSnowflake instance and destroys all data.
stop([keep_storage])	Stop the current FractalSnowflake instance and destroys all data.

Attributes Documentation

logfilename

Methods Documentation

client() → qcfractal.interface.client.FractalClient
Builds a client from this server.

Returns An active client connected to the server.

Return type FractalClient

get_address(*endpoint: Optional[str] = None*) → str
Obtains the full URI for a given function on the FractalServer.

Parameters **endpoint** (*Optional[str], optional*) – Specifies a endpoint to provide the URI for. If None returns the server address.

Returns The endpoint URI

Return type str

restart (*timeout: int = 5*) → None
Restarts the current FractalSnowflake instances and destroys all data in the process.

show_log (*nlines: int = 20*, *clean: bool = True*, *show: bool = True*)
Displays the FractalSnowflakes log data.

Parameters

- **nlines** (*int, optional*) – The the last n lines of the log.
- **clean** (*bool, optional*) – If True, cleans the log of manager operations where nothing happens.
- **show** (*bool, optional*) – If True prints to the log, otherwise returns the result text.

Returns Description

Return type TYPE

start (*timeout: int = 5*) → None
Stop the current FractalSnowflake instance and destroys all data.

stop (*keep_storage: bool = False*) → None
Stop the current FractalSnowflake instance and destroys all data.

Parameters **keep_storage** (*bool, optional*) – Does not delete the storage object if True.

PostgresHarness

class qcfractal.**PostgresHarness** (*config: Union[Dict[str, Any], qcfractal.config.FractalConfig], quiet: bool = True, logger: print = <built-in function print>*)
Bases: object

Methods Summary

alembic_commands()	
backup_database([filename])	
command(cmd[, check])	Runs psql commands and returns their output while connected to the correct postgres instance.
connect([database])	Builds a psycopg2 connection object.
create_database(database_name)	Creates a new database for the current postgres instance.
create_tables()	Create database tables using SQLAlchemy models
database_size()	Returns a pretty formatted string of the database size.
database_uri()	Provides the full PostgreSQL URI string.
init_database()	
initialize_postgres()	Initializes and starts the current postgres instance.
is_alive([database])	Checks if the postgres is alive, and optionally if the database is present.
logger(msg)	Prints a logging message depending on quiet settings.

continues on next page

Table 7 – continued from previous page

pg_ctl(cmds)	Runs pg_ctl commands and returns their output while connected to the correct postgres instance.
restore_database(filename)	
shutdown()	Shutsdown the current postgres instance.
start()	Starts a PostgreSQL server based off the current configuration parameters.
update_db_version()	Update current version of QCFractal in the DB
upgrade()	Upgrade the database schema using the latest alembic revision.

Methods Documentation

alembic_commands () → List[str]

backup_database (*filename: Optional[str] = None*) → None

command (*cmd: str*, *check: bool = True*) → Any
Runs psql commands and returns their output while connected to the correct postgres instance.

Parameters **cmd** (*str*) – A psql command string. Description

connect (*database: Optional[str] = None*) → Connection
Builds a psycopg2 connection object.

Parameters **database** (*Optional[str], optional*) – The database to connect to, otherwise defaults to None

Returns A live Connection object.

Return type Connection

create_database (*database_name: str*) → bool
Creates a new database for the current postgres instance. If the database is existing, no changes to the database are made.

Parameters **database_name** (*str*) – The name of the database to create.

Returns If the operation was successful or not.

Return type bool

create_tables ()
Create database tables using SQLAlchemy models

database_size () → str
Returns a pretty formatted string of the database size.

database_uri () → str
Provides the full PostgreSQL URI string.

Returns The database URI

Return type str

init_database () → None

initialize_postgres () → None
Initializes and starts the current postgres instance.

is_alive (*database: Optional[str] = None*) → bool
Checks if the postgres is alive, and optionally if the database is present.

Parameters **database** (*Optional[str], optional*) – The datbase to connect to

Returns If True, the postgres database is alive.

Return type bool

logger (*msg: str*) → None
Prints a logging message depending on quiet settings.

Parameters **msg** (*str*) – The message to show.

pg_ctl (*cmds: List[str]*) → Any
Runs pg_ctl commands and returns their output while connected to the correct postgres instance.

Parameters **cmds** (*List[str]*) – A list of PostgreSQL pg_ctl commands to run.

restore_database (*filename*) → None

shutdown () → Any
Shutsdown the current postgres instance.

start () → Any
Starts a PostgreSQL server based off the current configuration parameters. The server must be initialized and the configured port open.

update_db_version ()
Update current version of QCFractal in the DB

upgrade ()
Upgrade the database schema using the latest alembic revision. The database data won't be deleted.

QueueManager

class qcfractal.**QueueManager**(*client: FractalClient*, *queue_client: BaseAdapter*, *logger: Optional[logging.Logger] = None*, *max_tasks: int = 200*, *queue_tag: Optional[Union[str, List[str]]] = None*, *manager_name: str = 'unlabeled'*, *update_frequency: Union[int, float] = 2*, *verbose: bool = True*, *server_error_retries: Optional[int] = 1*, *stale_update_limit: Optional[int] = 10*, *cores_per_task: Optional[int] = None*, *memory_per_task: Optional[float] = None*, *nodes_per_task: Optional[int] = None*, *cores_per_rank: Optional[int] = 1*, *scratch_directory: Optional[str] = None*, *retries: Optional[int] = 2*, *configuration: Optional[Dict[str, Any]] = None*)

Bases: object

This object maintains a computational queue and watches for finished tasks for different queue backends. Finished tasks are added to the database and removed from the queue.

Variables

- **client** (*FractalClient*) – A FractalClient connected to a server.
- **queue_adapter** (*QueueAdapter*) – The DBAdapter class for queue abstraction
- **errors** (*dict*) – A dictionary of current errors
- **logger** (*logging.logger. Optional, Default: None*) – A logger for the QueueManager

Methods Summary

add_exit_callback(callback, *args, **kwargs)	Adds additional callbacks to perform when closing down the server.
assert_connected()	Raises an error for functions that require a server connection.
await_results()	A synchronous method for testing or small launches that awaits task completion.
close_adapter()	Closes down the underlying adapter.
connected()	Checks the connection to the server.
heartbeat()	Provides a heartbeat to the connected Server.
list_current_tasks()	Provides a list of tasks currently in the queue along with the associated keys.
name()	Returns the Managers full name.
shutdown()	Shutdown the manager and returns tasks to queue.
start()	Starts up all IOLoops and processes.
stop([signame, signum, stack])	Shuts down all IOLoops and periodic updates.
test([n])	Tests all known programs with simple inputs to check if the Adapter is correctly instantiated.
update([new_tasks, allow_shutdown])	Examines the queue for completed tasks and adds successful completions to the database while unsuccessful are logged for future inspection.

Methods Documentation

add_exit_callback (*callback: Callable*, **args: List[Any]*, ***kwargs: Dict[Any, Any]*) → None
Adds additional callbacks to perform when closing down the server.

Parameters

- **callback** (*callable*) – The function to call at exit
- ***args** – Arguments to call with the function.
- ****kwargs** – Kwargs to call with the function.

assert_connected () → None
Raises an error for functions that require a server connection.

await_results () → bool
A synchronous method for testing or small launches that awaits task completion.

Returns Return True if the operation completed successfully

Return type bool

close_adapter () → bool
Closes down the underlying adapter.

connected () → bool
Checks the connection to the server.

heartbeat () → None
Provides a heartbeat to the connected Server.

list_current_tasks () → List[Any]
Provides a list of tasks currently in the queue along with the associated keys.

Returns **ret** – All tasks currently still in the database

Return type list of tuples

name() → str
Returns the Managers full name.

shutdown() → Dict[str, Any]
Shutdown the manager and returns tasks to queue.

start() → None
Starts up all IOLoops and processes.

stop(*signame='Not provided'*, *signum=None*, *stack=None*) → None
Shuts down all IOLoops and periodic updates.

test(*n=1*) → bool
Tests all known programs with simple inputs to check if the Adapter is correctly instantiated.

update(*new_tasks: bool = True*, *allow_shutdown=True*) → bool
Examines the queue for completed tasks and adds successful completions to the database while unsuccessful are logged for future inspection.

Parameters

- **new_tasks** (*bool, optional, Default: True*) – Try to get new tasks from the server
- **allow_shutdown** (*bool, optional, Default: True*) – Allow function to attempt graceful shutdowns in the case of stale job or fatal error limits. Does not prevent errors from being raise, but mostly used to prevent infinite loops when update is called from *shutdown* itself

TemporaryPostgres

class qcfractal.**TemporaryPostgres**(*database_name: Optional[str] = None*, *tmpdir: Optional[str] = None*, *quiet: bool = True*, *logger: print = <built-in function print>*)

Bases: object

Methods Summary

database_uri([safe, database])	Provides the full Postgres URI string.
stop()	Shuts down the Snowflake instance.

Methods Documentation

database_uri(*safe: bool = True*, *database: Optional[str] = None*) → str
Provides the full Postgres URI string.

Parameters

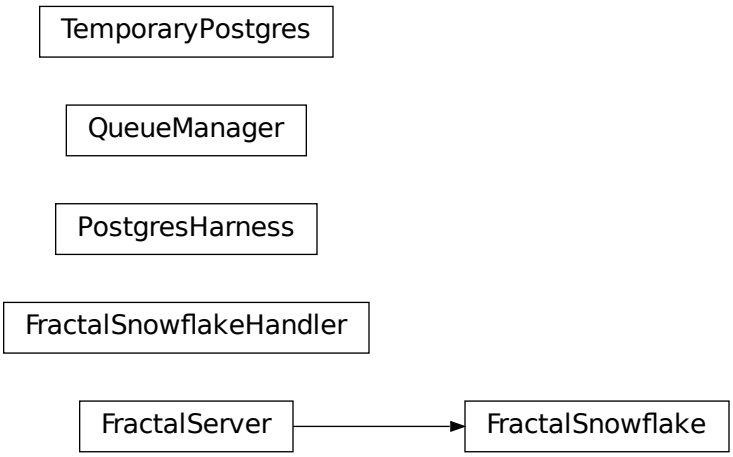
- **safe** (*bool, optional*) – If True, hides the postgres password.
- **database** (*Optional[str], optional*) – An optional database to add to the string.

Returns The database URI

Return type str

stop() → None
Shuts down the Snowflake instance. This instance is not recoverable after a stop call.

Class Inheritance Diagram



5.23.2 qcfractal.queue Package

Initializer for the queue_handler folder

Functions

build_queue_adapter(workflow_client[, logger])	Constructs a queue manager based off the incoming queue socket type.

build_queue_adapter

qcfractal.queue.**build_queue_adapter**(*workflow_client*, *logger=None*, ***kwargs*) → BaseAdapter
Constructs a queue manager based off the incoming queue socket type.

Parameters

- **workflow_client** (*object*) –

  A object wrapper for different distributed workflow types. The following input types are valid

  - Python Processes: "concurrent.futures.process.ProcessPoolExecutor"
  - Dask Distributed: "distributed.Client"

- Fireworks: "fireworks.LaunchPad"
- Parsl: "parsl.config.Config"

- **logger** (*logging.Logger, Optional. Default: None*) – Logger to report to
- ****kwargs** – Additional kwargs for the Adapter

Returns **ret** – Returns a valid Adapter for the selected computational queue

Return type Adapter

Classes

ComputeManagerHandler(application, request, ...)	Handles management/status querying of managers
QueueManager(client, queue_client, logger, ...)	This object maintains a computational queue and watches for finished tasks for different queue backends.
QueueManagerHandler(application, request, ...)	Manages the task queue.
ServiceQueueHandler(application, request, ...)	Handles service management (querying/add/modifying)
TaskQueueHandler(application, request, **kwargs)	Handles task management (querying/adding/modifying tasks)

ComputeManagerHandler

class qcfractal.queue.**ComputeManagerHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: qcfractal.web_handlers.APIHandler

Handles management/status querying of managers

Methods Summary

get()	Gets manager information from the task queue

Methods Documentation

get()

Gets manager information from the task queue

QueueManager

class qcfractal.queue.**QueueManager**(*client: FractalClient*, *queue_client: BaseAdapter*, *logger: Optional[logging.Logger] = None*, *max_tasks: int = 200*, *queue_tag: Optional[Union[str, List[str]]] = None*, *manager_name: str = 'unlabeled'*, *update_frequency: Union[int, float] = 2*, *verbose: bool = True*, *server_error_retries: Optional[int] = 1*, *stale_update_limit: Optional[int] = 10*, *cores_per_task: Optional[int] = None*, *memory_per_task: Optional[float] = None*, *nodes_per_task: Optional[int] = None*, *cores_per_rank: Optional[int] = 1*, *scratch_directory: Optional[str] = None*, *retries: Optional[int] = 2*, *configuration: Optional[Dict[str, Any]] = None*)

Bases: object

This object maintains a computational queue and watches for finished tasks for different queue backends. Finished tasks are added to the database and removed from the queue.

Variables

- **client** (*FractalClient*) – A FractalClient connected to a server.
- **queue_adapter** (*QueueAdapter*) – The DBAdapter class for queue abstraction
- **errors** (*dict*) – A dictionary of current errors
- **logger** (*logging.logger. Optional, Default: None*) – A logger for the QueueManager

Methods Summary

add_exit_callback(callback, *args, **kwargs)	Adds additional callbacks to perform when closing down the server.
assert_connected()	Raises an error for functions that require a server connection.
await_results()	A synchronous method for testing or small launches that awaits task completion.
close_adapter()	Closes down the underlying adapter.
connected()	Checks the connection to the server.
heartbeat()	Provides a heartbeat to the connected Server.
list_current_tasks()	Provides a list of tasks currently in the queue along with the associated keys.
name()	Returns the Managers full name.
shutdown()	Shutdown the manager and returns tasks to queue.
start()	Starts up all IOLoops and processes.
stop([signame, signum, stack])	Shuts down all IOLoops and periodic updates.
test([n])	Tests all known programs with simple inputs to check if the Adapter is correctly instantiated.
update([new_tasks, allow_shutdown])	Examines the queue for completed tasks and adds successful completions to the database while unsuccessful are logged for future inspection.

Methods Documentation

add_exit_callback(*callback: Callable*, **args: List[Any]*, ***kwargs: Dict[Any, Any]*) → None
Adds additional callbacks to perform when closing down the server.

Parameters

- **callback** (*callable*) – The function to call at exit
- ***args** – Arguments to call with the function.
- ****kwargs** – Kwargs to call with the function.

assert_connected() → None
Raises an error for functions that require a server connection.

await_results() → bool
A synchronous method for testing or small launches that awaits task completion.

Returns Return True if the operation completed successfully

Return type bool

close_adapter() → bool
Closes down the underlying adapter.

connected() → bool
Checks the connection to the server.

heartbeat() → None
Provides a heartbeat to the connected Server.

list_current_tasks() → List[Any]
Provides a list of tasks currently in the queue along with the associated keys.

Returns **ret** – All tasks currently still in the database

Return type list of tuples

name() → str
Returns the Managers full name.

shutdown() → Dict[str, Any]
Shutdown the manager and returns tasks to queue.

start() → None
Starts up all IOLoops and processes.

stop(*signame='Not provided'*, *signum=None*, *stack=None*) → None
Shuts down all IOLoops and periodic updates.

test(*n=1*) → bool
Tests all known programs with simple inputs to check if the Adapter is correctly instantiated.

update(*new_tasks: bool = True*, *allow_shutdown=True*) → bool
Examines the queue for completed tasks and adds successful completions to the database while unsuccessful are logged for future inspection.

Parameters

- **new_tasks** (*bool, optional, Default: True*) – Try to get new tasks from the server
- **allow_shutdown** (*bool, optional, Default: True*) – Allow function to attempt graceful shutdowns in the case of stale job or fatal error limits. Does not prevent errors from being raise, but mostly used to prevent infinite loops when update is called from *shutdown* itself

QueueManagerHandler

class qcfractal.queue.**QueueManagerHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: qcfractal.web_handlers.APIHandler

Manages the task queue.

Used by compute managers for getting tasks, posting completed tasks, etc.

Methods Summary

get()	Pulls new tasks from the task queue
insert_complete_tasks(storage_socket, body, ...)	
post()	Posts complete tasks to the task queue
put()	Various manager manipulation operations

Methods Documentation

get()
Pulls new tasks from the task queue

static insert_complete_tasks(*storage_socket*, *body*, *logger*)

post()
Posts complete tasks to the task queue

put()
Various manager manipulation operations

ServiceQueueHandler

class qcfractal.queue.**ServiceQueueHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Bases: qcfractal.web_handlers.APIHandler

Handles service management (querying/add/modifying)

Methods Summary

get()	Gets information about services from the service queue.
post()	Posts new services to the service queue.
put()	Modifies services in the service queue

Methods Documentation

get()
Gets information about services from the service queue.

post()
Posts new services to the service queue.

put()
Modifies services in the service queue

TaskQueueHandler

class qcfractal.queue.**TaskQueueHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)
Bases: qcfractal.web_handlers.APIHandler

Handles task management (querying/adding/modifying tasks)

Methods Summary

get()	Gets task information from the task queue
post()	Posts new tasks to the task queue.
put()	Modifies tasks in the task queue

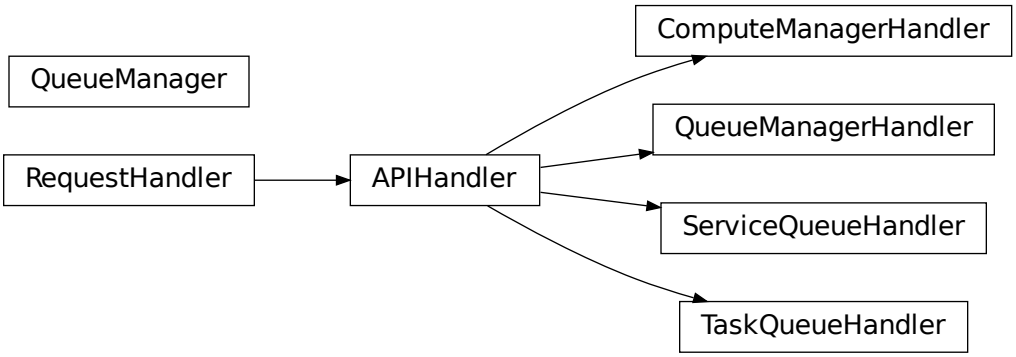
Methods Documentation

get()
Gets task information from the task queue

post()
Posts new tasks to the task queue.

put()
Modifies tasks in the task queue

Class Inheritance Diagram



5.23.3 qcfractal.services Package

Base import for services

Functions

construct_service(storage_socket, logger, data)	Initializes a service from a JSON blob.
initialize_service(storage_socket, logger, . . .)	Initializes a service from a API call.

construct_service

qcfractal.services.**construct_service**(*storage_socket*, *logger*, *data*)

Initializes a service from a JSON blob.

Parameters

- **storage_socket** (*StorageSocket*) – A StorageSocket to the currently active database
- **logger** – A logger for use by the service
- **data** (*dict*) – The associated JSON blob with the service

Returns Returns an instantiated service

Return type Service

initialize_service

qcfractal.services.**initialize_service**(*storage_socket*, *logger*, *service_input*, *tag=None*, *priority=None*)

Initializes a service from a API call.

Parameters

- **storage_socket** (*StorageSocket*) – A StorageSocket to the currently active database
- **logger** – A logger for use by the service
- **service_input** – The service to be initialized.
- **tag** (*Optional*) – Optional tag to user with the service. Defaults to None
- **priority** – The priority of the service.

Returns Returns an instantiated service

Return type Service

5.24 Database Design

Warning: Final MongoDB Supported Version: 0.7.0

0.7.0 is the last major release which support MongoDB. Fractal is moving towards a PostgreSQL database to make upgrades more stable and because it is more suited to the nature of QCArchive Data. The upgrade path from MongoDB to PostgreSQL will be provided by the Fractal developers in the next release. Due to the complex nature of the upgrade, the PostgreSQL upgrade will through scripts which will be provided. After the PostgreSQL upgrade, there will be built-in utilities to upgrade the Database.

QCArchive stores all its data and computations in a database in the backend of QCFractal. The DB is designed with extensibility in mind, allowing flexibility and easy accommodation of future features. The current backend of the DB storage is build on top of a non-relational DB, MongoDB, but it can be easily implemented in a Relational DB like MySQL or Postgres. In addition, Object Relational Mapping (ORM) is used to add some structure and ensure validation on the MongoDB which does not have any by definition. The ORM used is the most popular general MongoDB Python ORM, Mongoengine.

The main idea behind QCArchive DB design is to be able to store and retrieve wide range of Quantum Chemistry computations using different programs and variety of configurations. The DB also stores information about jobs submitted to request computations, and all their related data, along with registered users and computational managers.

QCArchive DB is organized into a set of tables (or documents), each of which are detailed below.

5.24.1 1) Molecule

The molecule table stores molecules used in any computation in the system. The molecule structure is based on the standard QCSchema. It stores entries like geometry, masses, and fragment charges. Please refer to the QCSchema for a complete description of all the possible fields.

5.24.2 2) Keyword

Keywords are a store of key-value pairs that are configuration for some QC program. It is flexible and there is no restriction on what configuration can be stored here. This table referenced by the `Result` table.

5.24.3 3) Result

This table stores the actual computation results along with the attributes used to calculate it. Each entry is a single unit of computation. The following are the unique set of keys (or indices) that define a result:

- `driver` - The type of calculation being evaluated (i.e. `energy`, `gradient`, `hessian`, `properties`)
- `program`: such as `gamess` or `psi4` (lower case)
- `molecule`: the ID of the molecule in the `Molecule` table
- `method`: the method used in the computation (b3lyp, mp2, ccsd(t))
- `keywords`: the ID of the keywords in the `Keywords` table
- `basis`: the name of the basis used in the computation (6-31g, cc-pvdz, def2-svp)

For more information see: *Results*.

5.24.4 4) Procedure

Procedures are also computational results but in a more complex fashion. They perform more aggregate computations like optimizations, torsion drive, and grid optimization. The DB can support new types of optimizations by inheriting from the the base procedure table. Each procedure usually reference several other results from the `Results` table, and possibly other procedures (self-reference).

5.24.5 5) Services

Services are more flexible workflows that eventually produce results to be stored in the `Result` and/or the `Procedure` tables when they are done. So, from the DB point of view, this is an intermediate table for on going iterative computations.

More about services in QCArchive can be found here: *Services*.

5.24.6 6) TaskQueue

This table is the main task queue of the system. Tasks are submitted to this table by QCFractal and wait for a manager to pull it for computation. Each task in the queue references a `Result` or a `Procedure`, meaning that it is corresponding to a specific Quantum computation. The table stores the status of the task (`WAITING`, `RUNNING`, `COMPLETE`, and `ERROR`) and also keeps track of the execution manager and the modification dates.

5.24.7 7) QueueManagers

Managers are the registered servers for computing tasks from the `TaskQueue`. This table keep information about the server such as the host, cluster, number of completed tasks, submissions, and failures.

The database only keeps track of what *Tasks* have been handed out to each *Manager* and maintains a heartbeat to ensure the *Manager* is still connected. More information about the configuration and execution of managers can be found here: *Fractal Queue Managers*.

5.25 Glossary

This glossary contains the common terms which appear over the entire Fractal project. There are other, specialized glossaries for components of Fractal which are linked below to help group terms together with their contextual docs. Some terms may appear in multiple glossaries, but will always have the same meaning, e.g. *Queue Adapter* and *Adapter*.

DB Index A DB Index (or Database Index) is a commonly queried field used to speed up searches in a *DB Table*.

DB Socket A DB Socket (or Database Socket) is the interface layer between standard Python queries and raw SQL or MongoDB query language.

DB Table A set of data inside the Database which has a common *ObjectId*. The `table` name follows SQL conventions which is also known as a `collection` in MongoDB.

Fractal Config Directory The directory where QCFractal Server and Database configuration files live. This is also the home of the Database itself in the default configuration. Default path is `~/.qca/qcfractal`

Hash Index A index that hashes the information contained in the object in a reproducible manner. This hash index is only used to find duplicates and should not be relied upon as it may change in the future.

Molecule A unique 3D representation of a molecule. Any changes to the protonation state, multiplicity, charge, fragments, coordinates, connectivity, isotope, or ghost atoms represent a change in the molecule.

ObjectId A ObjectId (or Database ID) is a unique ID for a given row (a document or entry) in the database that uniquely defines that particular row in a *DB Table*. These rows are automatically generated and will be different for every database, but outlines ways to reference other rows in the database quickly. A ObjectId is unique to a DB Table.

Procedures On-node computations, these can either be a single computation (energy, gradient, property, etc.) or a series of calculations such as a geometry optimization.

Queue Adapter The interface between QCFractal's internal queue representation and other queueing systems such as Dask or Fireworks. Also see the *Adapter* in the *Manager* glossary.

Services Iterative workflows where the required computations are distributed via the queue and then are processed on the server to acquire the next iteration of calculations.

5.25.1 Contextually Organized Glossaries

- *Queue Manager Glossary*

5.26 Development Guidelines

QCArchive developers adhere to a set of guidelines, both in the software stylistic guide, and in the outward conduct. We are working on codifying these in a clean list here, but early guides can be as follows

5.26.1 Software Development Guides

We openly encourage development of the QCArchive and all of its projects in public discourse through GitHub and the QCArchive Slack (Join our Slack group).

All changes should be proposed through a PR to the main projects from Forks of said projects.

For more details about the development cycle and guidelines, please see the DevTools Readme on GitHub for the project.

5.26.2 Personal Conduct Guides

Basic rule of thumb: Be respectful, welcome people, keep all interactions harassment-free.

Please see the full Code of Conduct on the project's GitHub page for more information.

5.27 Changelog

5.27.1 0.15.3 / 2021-03-15

This is a small release focused on some database migrations to improve performance. This should greatly improve performance of certain actions (particularly task submission) with large databases.

This release also drops support for python < 3.7

Client and managers should not need to be upgraded.

- (GH#663) Adds indices to base_result and molecule (improves ability to delete orphan kvstore)
- (GH#664) Adds indices to base_result and access_log (improves existing procedure lookup)

5.27.2 0.15.0 / 2020-11-11

This release is focused on bugfixes, and laying some foundation for larger changes to come.

New features

- (GH#636) Add ability to profile fractal instances
- (GH#642) Add (experimental!) qcexport code to devtools

Enhancements

- (GH#629) (Standard) Output of torsion drive service is now captured and stored in the procedure record
- (GH#631) Compress errors on server

Bug Fixes

- (GH#624) Lock task queue rows to prevent multiple managers requesting the same task
- (GH#626) Fix printing of client version during version check failure
- (GH#632) Fix ordering of initial/final molecule in torsion drives
- (GH#637) Fix inability to shutdown ProcessPoolExecutor workers
- (GH#638) Fix incorrect error in datasets
- (GH#641) Fix exception in web handler that was polluting log files

Miscellaneous

- (GH#633, GH#634, GH#635, GH#639) Miscellaneous cleanup and removal of unused database columns

5.27.3 0.14.0 / 2020-09-30

New Features

- (GH#597) Add ability to query managers
- (GH#612) Enabled compression of KVStore (generally, outputs)
- (GH#617) Ability to control level of logging via the command line
- (GH#620) Add ability to regenerate and modify tasks

Enhancements

- (GH#592 and GH#615) Improved performance of task retrieval of managers
- (GH#620) Improve performance of task submission, and add additional logging

Bug Fixes

- (GH#603) Fix error when running older computations missing 'protocols'
- (GH#617) Fix printing of base folder with the CLI

5.27.4 0.13.1 / 2020-02-18

New Features

- (GH#566) A `list_keywords` function was added to `Dataset`.

Enhancements

- (GH#547, GH#553) Miscellaneous documentation edits and improvements.
- (GH#556) Molecule queries filtered on molecular formula no longer depend on the order of elements.
- (GH#565) `query` method for `Datasets` now returns collected records.

Bug Fixes

- (GH#559) Fixed an issue where Docker images did not have qcfractal in their PATH.
- (GH#561) Fixed a bug that caused errors with pandas v1.0.
- (GH#564) Fixes a bug where optimization protocols were not respected during torsiondrives and grid optimizations.

5.27.5 0.13.0 / 2020-01-15

New Features

- (GH#541) Managers can now accept multiple tags. Tasks are pulled first in order of tag, then priority, then creation time.
- (GH#544) Adds backup/restore commands to the QCFractal CLI to allow for easier backup and restore options.

Enhancements

- (GH#507) Automatically adds collection molecules in chunks if more than the current limit needs to be submitted.
- (GH#515) Conda environments now correspond to docker images in all deployed cases.
- (GH#524) The `delete_collection` function was added to `qcportal.FractalClient`.
- (GH#530) Adds the ability to specify cores per rank for node-parallel tasks in QCEngine.
- (GH#507) Adds a formatting and lint check to CI during pull requests.
- (GH#535) Allows dftd3 to be computed for all stoichiometries rather than just defaults.
- (GH#542) Important: TaskRecord.base_result is now an ObjectId (int or str), and no more a `DBRef`. So, code that uses `my_task.base_result.id` should change to simply use `my_task.base_result`.

Bug Fixes

- (GH#506) Fixes repeated visualize calls where previously the visualize call would corrupt local state.
- (GH#521) Fixes an issue where ProcessPoolExecutor returned the incorrect number of currently running tasks.
- (GH#522) Fixes a bug where `ProcedureDataset.status()` failed for specifications where only a subset was computed.
- (GH#525) This PR fixes ENTRYPOINT of the qcarchive_worker_openff worker. (Conda and Docker are not friends.)
- (GH#532) Fixes a testing subprocess routine when coverage is enabled for coverage 5.0 breaking changes.
- (GH#543) Fixes a bug where `qcfractal-server` "start" before an "upgrade" prevented the "upgrade" command from correctly running.
- (GH#545) Fixed an issue in Dataset.get_records() that could occur when the optional arguments keywords and basis were not provided.

5.27.6 0.12.2 / 2019-12-07

Enhancements

- (GH#477) Removes 0.12.x xfails when connecting to the server.
- (GH#481) Expands Parsl Manager Adapter to include ALCF requirements.
- (GH#483) Dataset Views are now much faster to load in HDF5.
- (GH#488) Allows gzipped dataset views.
- (GH#490) Computes checksums on gzipped dataset views.
- (GH#542) `TaskRecord.base_result` is now an `ObjectId`, and no more a `DBRef`. So, code that uses `my_task.base_result.id` should change to simply be `my_task.base_result`.

Bug Fixes

- (GH#486) Fixes pydantic `__repr__` issues after update.
- (GH#492) Fixes error where `ReactionDataset` didn't allow a minimum number of n-body expansion to be added.
- (GH#493) Fixes an issue with `ReactionDataset.get_molecules` when a subset is present.
- (GH#494) Fixes an issue where queries with `limit=0` erroneously returned all results.
- (GH#496) TorsionDrive tests now avoid 90 degree angles with RDKit to avoid some linear issues in the force-field and make them more stable.
- (GH#497) `TorsionDrive.get_history` now works for extremely large (1000+) optimizations in the procedure.

5.27.7 0.12.1 / 2019-11-08

Enhancements

- (GH#472) Update to GitHub ISSUE templates.
- (GH#473) Server `/information` endpoint now contains the number of records for molecules, results, procedures, and collections.
- (GH#474) Dataset Views can now be of arbitrary shape.
- (GH#475) Changes the default formatting of the codebase to Black.

Bug Fixes

- (GH#470) Dataset fix for non-energy units.

5.27.8 0.12.0 / 2019-11-06

Highlights

- The ability to handle very large datasets (1M+ entries) quickly and efficiently.
- Store and compute Wavefunction information.
- Build, serve, and export views for Datasets that can stored in journal supplementary information or services like Zenodo.
- A new GUI dashboard to observe the current state of the server, see statistics, and fix issues.

New Features

- (GH#433 and GH#462) `Dataset` and `ReactionDataset` (`interface.collections`) now have a `download` method which downloads a frozen view of the dataset. This view is used to speed up calls to `get_values`, `get_molecules`, `get_entries`, and `list_values`.
- (GH#440) Wavefunctions can now be stored in the database using Result `protocols`.
- (GH#453) The server now periodically logs manager and current state to provide data over time.
- (GH#460) Contributed values are now in their own table to speed up access of Collections.
- (GH#461) Services now update their corresponding record every iteration. An example is a torsiondrive which now updates the `optimization_history` field each iteration.

Enhancements

- (GH#429) Enables protocols for `OptimizationDataset` collections.
- (GH#430) Adds additional QCPortal type hints.
- (GH#433, GH#443) `Dataset` and `ReactionDataset` (`interface.collections`) are now faster for calls to calls to `get_values`, `get_molecules`, `get_entries`, and `list_values` for large datasets if the server is configured to use frozen views. See "Server-side Dataset Views" documentation. Subsets may be passed to `get_values`, `get_molecules`, and `get_entries`
- (GH#447) Enables the creation of plaintext (xyz and csv) output from Dataset Collections.

- (GH#455) Projection queries should now be much faster as excluded results are not pulled to the server.
- (GH#458) Collections now have a metadata field.
- (GH#463) `FractalClient.list_collections` by default only returns collections whose visibility flag is set to true, and whose group is "default". This change was made to filter out in-progress, intermediate, and specialized collections.
- (GH#464) Molecule insert speeds are now 4-16x faster.

Bug Fixes

- (GH#424) Fixes a `ReactionDataset.visualize` bug with `groupby='D3'`.
- (GH#456, GH#452) Queries that project hybrid properties should now work as expected.

Deprecated Features

- (GH#426) In `Dataset` and `ReactionDataset` (`interface.collections`), the previously deprecated functions `query`, `get_history`, and `list_history` have been removed.

Optional Dependency Changes

- (GH#454) Users of the optional Parsl queue adapter are required to upgrade to Parsl v0.9.0, which fixes issues that caused SLURM managers to crash.

5.27.9 0.11.0 / 2019-10-01

New Features

- (GH#420) Pre-storage data handling through Elemental's `Protocols` feature are now present in Fractal. Although only optimization protocols are implemented functionally, the database side has been upgraded to store protocol settings.

Enhancements

- (GH#385, GH#404, GH#411) `Dataset` and `ReactionDataset` have five new functions for accessing data. `get_values` returns the canonical headline value for a dataset (e.g. the interaction energy for S22) in data columns with caching, both for result-backed values and contributed values. This function replaces the now-deprecated `get_history` and `get_contributed_values`. `list_values` returns the list of data columns available from `get_values`. This function replaces the now-deprecated `list_history` and `list_contributed_values`. `get_records` either returns `ResultRecord` or a projection. For the case of `ReactionDataset`, the results are broken down into component calculations. The function replaces the now-deprecated `query`. `list_records` returns the list of data columns available from `get_records`. `get_molecules` returns the `Molecule` associated with a dataset.
- (GH#393) A new feature added to `Client` to be able to have more custom and fast queries, the `custom_query` method. Those fast queries are now used in `torsiondrive.get_final_molecules` and `torsiondrive.get_final_results`. More Advanced queries will be added.
- (GH#394) Adds `tag` and `manager` selector fields to `client.query_tasks`. This is helpful for managing jobs in the queue and detecting failures.
- (GH#400, GH#401, GH#410) Adds Dockerfiles corresponding to builds on Docker Hub.

- (GH#406) The `Dataset` collection's primary indices (database level) have been updated to reflect its new understanding.

Bug Fixes

- (GH#396) Fixed a bug in internal `Dataset` function which caused `ComputeResponse` to be truncated when the number of calculations is larger than the query_limit.
- (GH#403) Fixed `Dataset.get_values` for any method which involved DFTD3.
- (GH#409) Fixed a compatibility bug in specific version of Intel-OpenMP by skipping version 2019.5-281.

Documentation Improvements

- (GH#399) A Kubernetes quickstart guide has been added.

5.27.10 0.10.0 / 2019-08-26

Note: Stable Beta Release

This release marks Fractal's official Stable Beta Release. This means that future, non-backwards compatible changes to the API will result in depreciation warnings.

Enhancements

- (GH#356) Collections' database representations have been improved to better support future upgrade paths.
- (GH#375) Dataset Records are now copied alongside the Collections.
- (GH#377) The `testing` suite from Fractal now exposes as a PyTest entry-point when Fractal is installed so that tests can be run from anywhere with the `--pyargs qcfractal` flag of `pytest`.
- (GH#384) "Dataset Records" and "Reaction Dataset Records" have been renamed to "Dataset Entry" and "Reaction Dataset Entry" respectively.
- (GH#387) The auto-documentation tech introduced in GH#321 has been replaced by the improved implementation in Elemental.

Bug Fixes

- (GH#388) Queue Manager shutdowns will now signal to reset any running tasks they own.

Documentation Improvements

- (GH#372, GH#376) Installation instructions have been updated and typo-corrected such that they are accurate now for both Conda and PyPi.

5.27.11 0.9.0 / 2019-08-16

New Features

- (GH#354) Fractal now takes advantage of Elemental's new Msgpack serialization option for Models. Serialization defaults to msgpack when available (`conda install msgpack-python [-c conda-forge]`), falling back to JSON otherwise. This results in substantial speedups for both serialization and deserialization actions and should be a transparent replacement for users within Fractal, Engine, and Elemental themselves.
- (GH#358) Fractal Server now exposes a CLI for user/permissions management through the `qcfractal-server user` command. See the full documentation for details.
- (GH#358) Fractal Server's CLI now supports user manipulations through the `qcfractal-server user` subcommand. This allows server administrators to control users and their access without directly interacting with the storage socket.

Enhancements

- (GH#330, GH#340, GH#348, GH#349) Many Pydantic based Models attributes are now documented and in an on-the-fly manner derived from the Pydantic Schema of those attributes.
- (GH#335) Dataset's `get_history` function is fixed by allowing the ability to force a new query even if one has already been cached.
- (GH#338) The Queue Manager which generated a `Result` is now stored in the `Result` records themselves.
- (GH#341) Skeletal Queue Manager YAML files can now be generated through the `--skel` or `--skeleton` CLI flag on `qcfractal-manager`
- (GH#361) Staged DB's in Fractal copy Alembic alongside them.
- (GH#363) A new REST API hook for services has been added so Clients can manage Services.

Bug Fixes

- (GH#359) A `FutureWarning` from Pandas has been addressed before it becomes an error.

Documentation Improvements

- (GH#351, GH#352, GH#353, GH#360, GH#362, GH#364, GH#366, GH#368) The documentation has been significantly edited to be up to date, fix numerous typos, reworded and refined for clarity, and overall flow better between pages.

5.27.12 0.8.0 / 2019-07-25

Breaking Changes

Warning: PostgreSQL is now the only supported database backend.

Fractal has officially dropped support for MongoDB in favor of PostgreSQL as our database backend. Although MongoDB served the start of Fractal well, our database design as evolved since then and will be better served by PostgreSQL.

New Features

- (GH#307, GH#319 GH#321) Fractal's Server CLI has been overhauled to more intuitively and intelligently control Server creation, startup, configuration, and upgrade paths. This is mainly reflected in a Fractal Server config file, a config folder (default location `~/.qca`, and sub-commands `init`, `start`, `config`, and `upgrade` of the `qcfractal-server (command)` CLI. See the full documentation for details
- (GH#323) First implementation of the `GridOptimizationDataset` for collecting Grid Optimization calculations. Not yet fully featured, but operational for users to start working with.

Enhancements

- (GH#291) Tests have been formally added for the Queue Manager to reduce bugs in the future. They cannot test on actual Schedulers yet, but its a step in the right direction.
- (GH#295) Quality of life improvement for Mangers which by default will be less noisy about heartbeats and trigger a heartbeat less frequently. Both options can still be controlled through verbosity and a config setting.
- (GH#296) Services are now prioritized by the date they are created to properly order the compute queue.
- (GH#301) `TorsionDriveDataset` status can now be checked through the `.status()` method which shows the current progress of the computed data.
- (GH#310) The Client can now modify tasks and restart them if need be in the event of random failures.
- (GH#313) Queue Managers now have more detailed statistics about failure rates, and core-hours consumed (estimated)
- (GH#314) The `PostgresHarness` has been improved to include better error handling if Postgress is not found, and will not try to stop/start if the target data directory is already configured and running.
- (GH#318) Large collections are now automatically paginated to improve Server/Client response time and reduce query sizes. See also GH#322 for the Client-side requested pagination.
- (GH#322) Client's can request paginated queries for quicker responses. See also GH#318 for the Server-side auto-pagination.
- (GH#322) `Record` models and their derivatives now have a `get_molecule()` method for fetching the molecule directly.
- (GH#324) Optimization queries for its trajectory pull the entire trajectory in one go and keep the correct order. `get_trajectory` also pulls the correct order.
- (GH#325) Collections' have been improved to be more efficient. Previous queries are cached locally and the `compute` call is now a single function, removing the need to make a separate call to the submission formation.
- (GH#326) `ReactionDataset` now explicitly groups the fragments to future-proof this method from upstream changes to `Molecule` fragmentation.

- (GH#329) All API requests are now logged server side anonymously.
- (GH#331) Queue Manager jobs can now auto-retry failed jobs a finite number of times through QCEngine's retry capabilities. This will only catch RandomErrors and all other errors are raised normally.
- (GH#332) SQLAlchemy layer on the PostgreSQL database has received significant polish

Bug Fixes

- (GH#291) Queue Manager documentation generation works on Pydantic 0.28+. A number as-of-yet uncaught/unseen bugs were revealed in tests and have been fixed as well.
- (GH#300) Errors thrown in the level between Managers and their Adapters now correctly return a `FailedOperation` instead of `dict` to be consistent with all other errors and not crash the Manager.
- (GH#301) Invalid passwords present a helpful error message now instead of raising an Internal Server Error to the user.
- (GH#306) The Manager CLI option `tasks-per-worker` is correctly hyphens instead of underscores to be consistent with all other flags.
- (GH#316) Queue Manager workarounds for older versions of Dask-Jobqueue and Parsl have been removed and implicit dependency on the newer versions of those Adapters is enforced on CLI usage of `qcfractal-manager`. These packages are *not required* for Fractal, so their versions are only checked when specifically used in the Managers.
- (GH#320) Duplicated `initial_molecules` in the `TorsionDriveDataset` will no longer cause a failure in adding them to the database while still preserving de-duplication.
- (GH#327) Jupyter Notebook syntax highlighting has been fixed on Fractal's documentation pages.
- (GH#331) The BaseModel/Settings auto-documentation function can no longer throw an error which prevents using the code.

Deprecated Features

- (GH#291) Queue Manager Template Generator CLI has been removed as its functionality is superseded by the `qcfractal-manager` CLI.

5.27.13 0.7.2 / 2019-05-31

New Features

- (GH#279) Tasks will be deleted from the `TaskQueue` once they are completed successfully.
- (GH#271) A new set of scripts have been created to facilitate migration between MongoDB and PostgreSQL.

Enhancements

- (GH#275) Documentation has been further updated to be more contiguous between pages.
- (GH#276) Imports and type hints in Database objects have been improved to remove ambiguity and make imports easier to follow.
- (GH#280) Optimizations queried in the database are done with a more efficient lazy `selectin`. This should make queries much faster.
- (GH#281) Database Migration tech has been moved to their own folder to keep them isolated from normal production code. This PR also called the testing database `test_qcarchivedb` to avoid clashes with production DBs. Finally, a new keyword for testing geometry optimizations has been added.

Bug Fixes

- (GH#280) Fixed a SQL query where `join` was set instead of `noload` in the lazy reference.
- (GH#283) The monkey-patch for Dask + LSF had a typo in the keyword for its invoke. This has been fixed for the monkey-patch, as the upstream change was already fixed.

5.27.14 0.7.1 / 2019-05-28

Bug Fixes

- (GH#277) A more informative error is thrown when Mongo is not found by `FractalSnowflake`.
- (GH#277) ID's are no longer presented when listing Collections in Portal to minimize extra data.
- (GH#278) Fixed a bug in Portal where the Server was not reporting the correct unit.

5.27.15 0.7.0 / 2019-05-27

> **Warning:** Final MongoDB Supported Release
>
> **This is the last major release which support MongoDB.** Fractal is moving towards a PostgreSQL for database to make upgrades more stable and because it is more suited to the nature of QCArchive Data. The upgrade path from MongoDB to PostgreSQL will be provided by the Fractal developers in the next release. Due to the complex nature of the upgrade, the PostgreSQL upgrade will through scripts which will be provided. After the PostgreSQL upgrade, there will be built-in utilities to upgrade the Database.

New Features

- (GH#206, GH#249, GH#264, GH#267) SQL Database is now feature complete and implemented. As final testing in production is continued, MongoDB will be phased out in the future.
- (GH#242) Parsl can now be used as an `Adapter` in the Queue Managers.
- (GH#247) The new `OptimizationDataset` collection has been added! This collection returns a set of optimized molecular structures given an initial input.
- (GH#254) The QCFractal Server Dashboard is now available through a Dash interface. Although not fully featured yet, future updates will improve this as features are requested.

- (GH#260) Its now even easier to install Fractal/Portal through conda with pre-built environments on the `qcarchive` conda channel. This channel only provides environment files, no packages (and there are not plans to do so.)
- (GH#269) The Fractal Snowflake project has been extended to work in Jupyter Notebooks. A Fractal Snowflake can be created with the `FractalSnowflakeHandler` inside of a Jupyter Session.

Database Compatibility Updates

- (GH#256) API calls to Elemental 0.4 have been updated. This changes the hashing system and so upgrading your Fractal Server instance to this (or higher) will require an upgrade path to the indices.

Enhancements

- (GH#238) `GridOptimizationRecord` supports the helper function `get_final_molecules` which returns the set of molecules at each final, optimized grid point.
- (GH#259) Both `GridOptimizationRecord` and `TorsionDriveRecord` support the helper function `get_final_results`, which is like `get_final_molecules`, but for x
- (GH#241) The visualization suite with Plotly has been made more general so it can be invoked in different classes. This particular PR updates the TorsionDriveDataSet objects.
- (GH#243) TorsionDrives in Fractal now support the updated Torsion Drive API from the underlying package. This includes both the new arguments and the "extra constraints" features.
- (GH#244) Tasks which fail are now more verbose in the log as to why they failed. This is additional information on top of the number of pass/fail.
- (GH#246) Queue Manager `verbosity` level is now passed down into the adapter programs as well and the log file (if set) will continue to print to the terminal as well as the physical file.
- (GH#247) Procedure classes now all derive from a common base class to be more consistent with one another and for any new Procedures going forward.
- (GH#248) Jobs which fail, or cannot be returned correctly, from Queue Managers are now better handled in the Manager and don't sit in the Manager's internal buffer. They will attempt to be returned to the Server on later updates. If too many jobs become stale, the Manager will shut itself down for safety.
- (GH#258 and GH#268) Fractal Queue Managers are now fully documented, both from the CLI and through the doc pages themselves. There have also been a few variables renamed and moved to be more clear the nature of what they do. See the PR for the renamed variables.
- (GH#251) The Fractal Server now reports valid minimum/maximum allowed client versions. The Portal Client will try check these numbers against itself and fail to connect if it is not within the Server's allowed ranges. Clients started from Fractal's `interface` do not make this check.

Bug Fixes

- (GH#248) Fixed a bug in Queue Managers where the extra worker startup commands for the Dask Adapter were not being parsed correctly.
- (GH#250) Record objects now correctly set their provenance time on object creation, not module import.
- (GH#253) A spelling bug was fixed in GridOptimization which caused hashing to not be processed correctly.
- (GH#270) LSF clusters not in `MB` for the units on memory by config are now auto-detected (or manually set) without large workarounds in the YAML file and the CLI file itself. Supports documented settings of LSF 9.1.3.

5.27.16 0.6.0 / 2019-03-30

Enhancements

- (GH#236 and GH#237) A large number of docstrings have been improved to be both more uniform, complete, and correct.
- (GH#239) DFT-D3 can now be queried through the `Dataset` and `ReactionDataset`.
- (GH#239) `list_collections` now returns Pandas Dataframes.

5.27.17 0.5.5 / 2019-03-26

New Features

- (GH#228) ReactionDatasets visualization statistics plots can now be generated through Plotly! This feature includes bar plots and violin plots and is designed for interactive use through websites, Jupyter notebooks, and more.
- (GH#233) TorsionDrive Datasets have custom visualization statistics through Plotly! This allows plotting 1-D torsion scans against other ones.

Enhancements

- (GH#226) LSF can now be specified for the Queue Managers for Dask Managers.
- (GH#228) Plotly is an optional dependency overall, it is not required to run QCFractal or QCPortal but will be downloaded in some situations. If you don't have Plotly installed, more graceful errors beyond just raw `ImportErrors` are given.
- (GH#234) Queue Managers now report the number of passed and failed jobs they return to the server and can also have verbose (debug level) outputs to the log.
- (GH#234) Dask-driven Queue Managers can now be set to simply scale up to a fixed number of workers instead of trying to adapt the number of workers on the fly.

Bug Fixes

- (GH#227) SGE Clusters specified in Queue Manager under Dask correctly process `job_extra` for additional scheduler headers. This is implemented in a stable way such that if the upstream Dask Jobqueue implements a fix, the Manager will keep working without needing to get a new release.
- (GH#234) Fireworks managers now return the same pydantic models as every other manager instead of raw dictionaries.

5.27.18 0.5.4 / 2019-03-21

New Features

- (GH#216) Jobs submitted to the queue can now be assigned a priority to be served out to the Managers.
- (GH#219) Temporary, pop-up, local instances of `FractalServer` can now be created through the `FractalSnowflake`. This creates an instance of `FractalServer`, with its database structure, which is entirely held in temporary storage and memory, all of which is deleted upon exit/stop. This feature is designed for those who want to tinker with Fractal without needed to create their own database or connect to a production `FractalServer`.
- (GH#220) Queue Managers can now set the `scratch_directory` variable that is passed to QCEngine and its workers.

Enhancements

- (GH#216) Queue Managers now report what programs and procedures they have access to and will only pull jobs they think they can execute.
- (GH#222) All of `FractalClient`'s methods now have full docstrings and type annotations for clairy
- (GH#222) Massive overhaul to the REST interface to simplify internal calls from the client and server side.
- (GH#223) `TorsionDriveDataset` objects are modeled through pydantic objects to allow easier interface with the database back end and data validation.

Bug Fixes

- (GH#215) Dask Jobqueue for the `qcfractal-manager` is now tested and working. This resolve the outstanding issue introduced in GH#211 and pushed in v0.5.3.
- (GH#216) Tasks are now stored as `TaskRecord` pydantic objects which now preempts a bug introduced from providing the wrong schema.
- (GH#217) Standalone QCPortal installs now report the correct version
- (GH#221) Fixed a bug in `ReactionDataset.query` where passing in `None` was treated as a string.

5.27.19 0.5.3 / 2019-03-13

New Features

- (GH#207) All compute operations can now be augmented with a `tag` which can be later consumed by different `QueueManagers` to only carry out computations with specified tags.
- (GH#210) Passwords in the database can now be generated for new users and user information can be updated (server-side only)
- (GH#210) `Collections` can now be updated automatically from the defaults
- (GH#211) The `qcfractal-manager` CLI command now accepts a config file for more complex managers through Dask JobQueue. As such, many of the command line flags have been altered and can be used to either spin up a PoolExecutor, or overwrite the config file on-the-fly. As of this PR, the Dask Jobqueue component has been untested. Future updates will indicate when this has been tested.

Enhancements

- (GH#203) `FractalClient`'s `get_X` methods have been renamed to `query_X` to better reflect what they actually do. An exception to this is the `get_collections` method which is still a true `get`.
- (GH#207) `FractalClient.list_collections` now respects show case sensitive results and queries are case insensitive
- (GH#207) `FractalServer` can now compress responses to reduce the amount of data transmitted over the serialization. The main benefactor here is the `OpenFFWorkflow` collection which has significant transfer speed improvements due to compression.
- (GH#207) The `OpenFFWorkflow` collection now has better validation on input and output data.
- (GH#210) The `OpenFFWorkflow` collection only stores database `id` to reduce duplication and data transfer quantities. This results in about a 50x duplication reduction.
- (GH#211) The `qcfractal-template` command now has fields for Fractal username and password.
- (GH#212) The docs for QCFractal and QCPortal have been split into separate structures. They will be hosted on separate (although linked) pages, but their content will all be kept in the QCFractal source code. QCPortal's docs are for most users whereas QCFractal docs will be for those creating their own Managers, Fractal instances, and developers.

Bug Fixes

- (GH#207) `FractalClient.get_collections` is now correctly case insensitive.
- (GH#210) Fixed a bug in the `iterate` method of services which returned the wrong status if everything completed right away.
- (GH#210) The `repr` of the MongoEngine Socket now displays correctly instead of crashing the socket due to missing attribute

5.27.20 0.5.2 / 2019-03-08

New Features

- (GH#197) New `FractalClient` instances will automatically connect to the central MolSSI Fractal Server

Enhancements

- (GH#195) Read-only access has been granted to many objects separate from their write access. This is in contrast to the previous model where either there was no access security, or everything was access secure.
- (GH#197) Unknown stoichiometry are no longer allowed in the `ReactionDataset`
- (GH#197) CLI for FractalServer uses Executor only to encourage using the Template Generator introduced in GH#177.
- (GH#197) `Dataset` objects can now query keywords from aliases as well.

Bug Fixes

- (GH#195) Manager cannot pull too many tasks and potentially loose data due to query limits.
- (GH#195) `Records` now correctly adds Provenance information
- (GH#196) `compute_torsion` example update to reflect API changes
- (GH#197) Fixed an issue where CLI input flags were not correctly overwriting default values
- (GH#197) Fixed an issue where `Collections` were not correctly updating when the `save` function was called on existing objects in the database.
- (GH#197) `_qcfractal_tags` are no longer carried through the `Records` objects in errant.
- (GH#197) Stoichiometry information is no longer accepted in the `Dataset` object since this is not used in this class of object anymore (see `ReactionDataset`).

5.27.21 0.5.1 / 2019-03-04

New Features

- (GH#177) Adds a new `qcfractal-template` command to generate `qcfractal-manager` scripts.
- (GH#181) Pagination is added to queries, defaults to 1000 matches.
- (GH#185) Begins setup documentation.
- (GH#186) Begins database design documentation.
- (GH#187) Results add/update is now simplified to always store entire objects rather than update partials.
- (GH#189) All database compute records now go through a single `BaseRecord` class that validates and hashes the objects.

Enhancements

- (GH#175) Refactors query massaging logic to a single function, ensures all program queries are lowercase, etc.
- (GH#175) Keywords are now lazy reference fields.
- (GH#182) Reworks models to have strict fields, and centralizes object hashing with many tests.
- (GH#183) Centralizes duplicate checking so that accidental mixed case duplicate results could go through.
- (GH#190) Adds QCArchive sphinx theme to the documentation.

Bug Fixes

- (GH#176) Benchmarks folder no longer shipped with package

5.27.22 0.5.0 / 2019-02-20

New Features

- (GH#165) Separates datasets into a Dataset, ReactionDataset, and OptimizationDataset for future flexability.
- (GH#168) Services now save their Procedure stubs automatically, the same as normal Procedures.
- (GH#169) `setup.py` now uses the README.md and conveys Markdown to PyPI.
- (GH#171) Molecule addition now takes in a flat list and returns a flat list of IDs rather than using a dictionary.
- (GH#173) Services now return their correspond Procedure ID fields.

Enhancements

- (GH#163) Ignores pre-existing IDs during storage add operations.
- (GH#167) Allows empty queries to successfully return all results rather than all data in a collection.
- (GH#172) Bumps pydantic version to 0.20 and updates API.

Bug Fixes

- (GH#170) Switches Parsl from IPPExecutor to ThreadExecutor to prevent some bad semaphore conflicts with PyTest.

5.27.23 0.5.0rc1 / 2019-02-15

New Features

- (GH#114) A new Collection: `Generic`, has been added to allow semi-structured user defined data to be built without relying only on implemented collections.
- (GH#125) QCElemental common pydantic models have been integrated throughout the QCFractal code base, making a common model repository for the prevalent `Molecule` object (and others) come from a single source. Also converted QCFractal to pass serialized pydantic objects between QCFractal and QCEngine to allow validation and (de)serialization of objects automatically.
- (GH#130, GH#142, and GH#145) Pydantic serialization has been added to all REST calls leaving and entering both QCFractal Servers and QCFractal Portals. This allows automatic REST call validation and formatting on both server and client sides.
- (GH#141 and GH#152) A new GridOptimizationRecord service has been added to QCFractal. This feature supports relative starting positions from the input molecule.

Enhancements

General note: `Options` objects have been renamed to `KeywordSet` to better match their goal (See GH#155.)

- (GH#110) QCFractal now depends on QCElemental and QCEngine to improve consistent imports.
- (GH#116) Queue Manger Adapters are now more generalized and inherit more from the base classes.
- (GH#118) Single and Optimization procedures have been streamlined to have simpler submission specifications and less redundancy.
- (GH#133) Fractal Server and Queue Manager startups are much more verbose and include version information.
- (GH#135) The TorsionDriveService has a much more regular structure based on pydantic models and a new TorsionDrive model has been created to enforce both validation and regularity.
- (GH#143) `Task``s in the Mongo database can now be referenced by multiple ``Results` and `Procedures` (i.e. a single `Result` or `Procedure` does not have ownership of a `Task`.)
- (GH#147) Service submission has been overhauled such that all services submit to a single source. Right now, only one service can be submitted at a time (to be expanded in a future feature.) TorsionDrive can now have multiple molecule inputs.
- (GH#149) Package import logic has been reworked to reduce the boot-up time of QCFractal from 3000ms at the worst to about 600ms.
- (GH#150) `KeywordSet` objects are now modeled much more consistently through pydantic models and are consistently hashed to survive round trip serialization.
- (GH#153) Datasets now support option aliases which map to the consistent `KeywordSet` models from GH#150.
- (GH#155) Adding multiple `Molecule` or `Result` objects to the database at the same time now always return their Database ID's if added, and order of returned list of ID's matches input order. This PR also renamed `Options` to `KeywordSet` to properly reflect the goal of the object.
- (GH#156) Memory and Number of Cores per Task can be specified when spinning up a Queue Manager and/or Queue Adapter objects. These settings are passed on to QCEngine. These must be hard-set by users and no environment inspection is done. Users may continue to choose not to set these and QCEngine will consume everything it can when it lands on a compute.
- (GH#162) Services can now be saved and fetched from the database through MongoEngine with document validation on both actions.

Bug Fixes

- (GH#132) Fixed MongoEngine Socket bug where calling some functions before others resulted in an error due to lack of initialized variables.
- (GH#133) `Molecule` objects cannot be oriented once they enter the QCFractal ecosystem (after optional initial orientation.) `Molecule` objects also cannot be oriented by programs invoked by the QCFractal ecosystem so orientation is preserved post-calculation.
- (GH#146) CI environments have been simplified to make maintaining them easier, improve test coverage, and find more bugs.
- (GH#158) Database addition documents in general will strip IDs from the input dictionary which caused issues from MongoEngine having a special treatment for the dictionary key "id".

5.27.24 0.4.0a / 2019-01-15

This is the fourth alpha release of QCFractal focusing on the database backend and compute manager enhancements.

New Features

- (GH#78) Migrates Mongo backend to MongoEngine.
- (GH#78) Overhauls tasks so that results or procedures own a task and ID.
- (GH#78) Results and procedures are now inserted upon creation, not just completion. Added a status field to results and procedures.
- (GH#78) Overhauls storage API to no longer accept arbitrary JSON queries, but now pinned kwargs.
- (GH#106) Compute managers now have heartbeats and tasks are recycled after a manager has not been heard from after a preset interval.
- (GH#106) Managers now also quietly shutdown on SIGTERM as well as SIGINT.

Bug Fixes

- (GH#102) Py37 fix for pydantic and better None defaults for `options`.
- (GH#107) `FractalClient.get_collections` now raises an exception when no collection is found.

5.27.25 0.3.0a / 2018-11-02

This is the third alpha release of QCFractal focusing on a command line interface and the ability to have multiple queues interacting with a central server.

New Features

- (GH#72) Queues are no longer required of FractalServer instances, now separate QueueManager instances can be created that push and pull tasks to the server.
- (GH#80) A Parsl Queue Manager was written.
- (GH#75) CLI's have been added for the `qcfractal-server` and `qcfractal-manager` instances.
- (GH#83) The status of server tasks and services can now be queried from a FractalClient.
- (GH#82) OpenFF Workflows can now add single optimizations for fragments.

Enhancements

- (GH#74) The documentation now has flowcharts showing task and service pathways through the code.
- (GH#73) Collection `.data` attributes are now typed and validated with pydantic.
- (GH#85) The CLI has been enhanced to cover additional features such as `queue-manager` ping time.
- (GH#84) QCEngine 0.4.0 and geomeTRIC 0.9.1 versions are now compatible with QCFractal.

Bug Fixes

- (GH#92) Fixes an error with query OpenFFWorkflows.

5.27.26 0.2.0a / 2018-10-02

This is the second alpha release of QCFractal containing architectural changes to the relational pieces of the database. Base functionality has been expanded to generalize the collection idea with BioFragment and OpenFFWorkflow collections.

Documentation

- (GH#58) A overview of the QCArchive project was added to demonstrate how all modules connect together.

New Features

- (GH#57) OpenFFWorkflow and BioFragment collections to support OpenFF uses cases.
- (GH#57) Requested compute will now return the id of the new submissions or the id of the completed results if duplicates are submitted.
- (GH#67) The OpenFFWorkflow collection now supports querying of individual geometry optimization trajectories and associated data for each torsiondrive.

Enhancements

- (GH#43) Services and Procedures now exist in the same unified table when complete as a single procedure can be completed in either capacity.
- (GH#44) The backend database was renamed to storage to prevent misunderstanding of the Database collection.
- (GH#47) Tests can that require an activate Mongo instance are now correctly skipped.
- (GH#51) The queue now uses a fast hash index to determine uniqueness and prevent duplicate tasks.
- (GH#52) QCFractal examples are now tested via CI.
- (GH#53) The MongoSocket `get_generic_by_id` was deprecated in favor of `get_generic` where an ID can be a search field.
- (GH#61, GH#64) TorsionDrive now tracks tasks via ID rather than hash to ensure integrity.
- (GH#63) The Database collection was renamed Dataset to more correctly illuminate its purpose.
- (GH#65) Collection can now be aquired directly from a client via the `client.get_collection` function.

Bug Fixes

- (GH#52) The molecular comparison technology would occasionally incorrectly orientate molecules.

5.27.27 0.1.0a / 2018-09-04

This is the first alpha release of QCFractal containing the primary structure of the project and base functionality.

New Features

- (GH#41) Molecules can now be queried by molecule formula
- (GH#39) The server can now use SSL protection and auto-generates SSL certificates if no certificates are provided.
- (GH#31) Adds authentication to the FractalServer instance.
- (GH#26) Adds TorsionDrive (formally Crank) as the first service.
- (GH#26) Adds a "services" feature which can create large-scale iterative workflows.
- (GH#21) QCFractal now maintains its own internal queue and uses queuing services such as Fireworks or Dask only for the currently running tasks

Enhancements

- (GH#40) Examples can now be testing through PyTest.
- (GH#38) First major documentation pass.
- (GH#37) Canonicalizes string formatting to the `"{}".format` usage.
- (GH#36) Fireworks workflows are now cleared once complete to keep the active entries small.
- (GH#35) The "database" table can now be updated so that database entries can now evolve over time.
- (GH#32) TorsionDrive services now track all computations that are completed rather than just the last iteration.
- (GH#30) Creates a Slack Community and auto-invite badge on the main readme.
- (GH#24) Remove conda-forge from conda-envs so that more base libraries can be used.

Bug Fixes

- Innumerable bug fixes and improvements in this alpha release.

PYTHON MODULE INDEX

q

INDEX

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W